3DO M2 Command List Toolkit

Version 2.7 – May 1996

Contents

4
Destination Blender

A
Register Setups

B
Sample Code

C
Function Calls

List of Figures

List of Tables

Preface

About This Book

This book describes the Command List Toolkit and it's relationship to the M2's 3D hardware rendering engine.

About the Audience

This manual is written for title developers. The information presented and the level of description is based on the assumption that you are familiar with both the C programming language, and three-dimensional graphics.

How The Command List Toolkit Programmer's Guide Is Organized

Chapter 1, The Command List Toolkit *–Provides a brief overview of the M2 3D hardware rendering engine, an introduction to the Command List Toolkit, and descriptions of the associated macros.*

Chapter 2, The Graphics State (GState) *–Describes the GState object and how it manages the command list buffers.*

Chapter 3, Textures *-Provides an overview of the M2's texture mapping logic, as well as descriptions of all of the applicable registers.*

Chapter 4, Destination Blender*–Provides an overview of the M2's destination blending logic, as well as descriptions of all of the applicable registers.*

Appendix A, Register Setups *-Provides the register setups for a variety of operations.*

Appendix B, Sample Code *-Provides a sample program.*

Appendix C, Function Calls *-Describes each of the Command List Toolkit function calls.*

Related Documentation

The following documents are recommended as a source of additional information:

- Foley, van Dam, Feiner, and Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley, 1990
- 3DO M2 Graphics Programmer's Guide

Chapter 1

The Command List Toolkit

The M2's 3D hardware rendering engine (the Triangle Engine or TE) renders graphics by reading blocks of data, called *command lists*, from memory and executing the instructions found in these blocks. The Command List Toolkit (CLT) is a collection of routines and macros that allow you to build these command lists easily and efficiently. Applications use the CLT to build commands for the Triangle Engine. These commands are built up into command list buffers, then handed off to the Triangle Engine's device driver.

The TE executes the commands, found in the command list buffers, asynchronously. So, while the Triangle Engine works on one block of command lists, the CPU is free to begin work on preparing the next set of Triangle Engine commands.

The main job of the CLT is to assign a set of names to all of the Triangle Engine registers and commands. These names are provided to the user through a header file. In addition, several shortcuts for using these names are provided in the form of C preprocessor macros.

This chapter describes the macros provided by the CLT. Your application can use these macros to communicate, at the most rudimentary level, with the Triangle Engine.

This chapter contains the following topics:

Introduction

Before we talk about the Command List Toolkit, let's take a brief look at how the Triangle Engine works. The TE is primarily composed of two groups of logic: the Texture Mapper and the Destination Blender. illustrates the data flow through the Triangle Engine. See Chapters 3 and 4 for more information about the Texture Mapper and the Destination Blender.

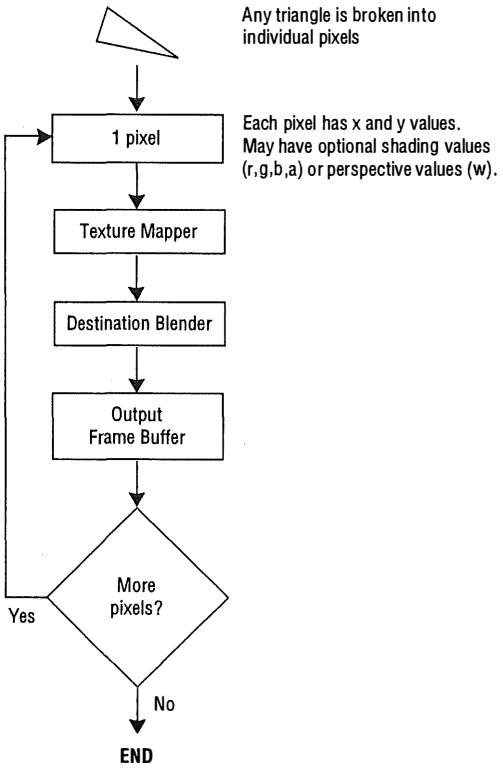


Figure 1-1 *Triangle Engine data flow.*

The TE renders triangles by first breaking them down into individual pixels. Each pixel has a set of *xy* coordinates that identify it, and may have optional values that define its shading or perspective.

The pixel goes first to the Texture Mapper. The texture mapper logic can render portions of a 2D image onto a 3D triangle, and it is here where texture and color are applied to the pixel (see Figure 1-2).

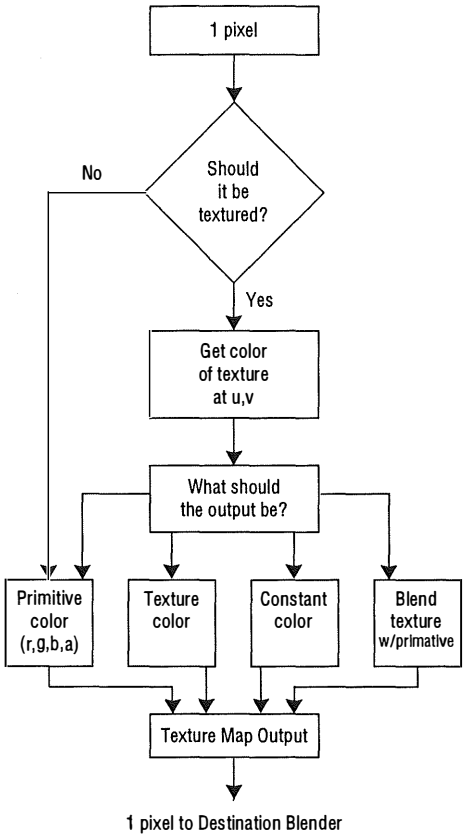


Figure 1-2 *Texture Mapper data flow.*

Next stop is the Destination Blender. The destination blender logic takes the pixel's texture information, and combines (blends) it with data from a number of sources. These sources can include the current Source frame buffer, a variety of constants and the control registers (see Figure 1-3).

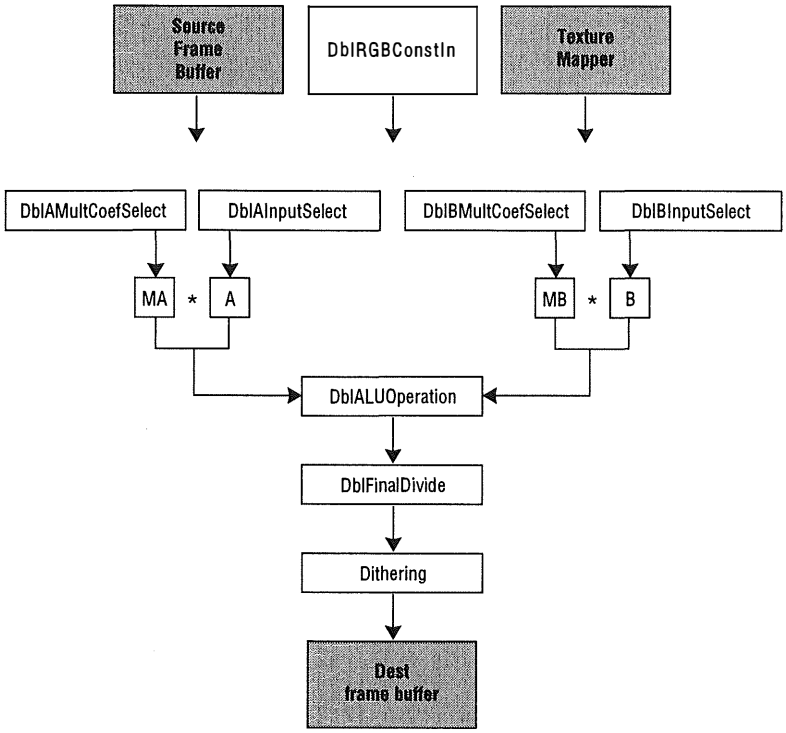


Figure 1-3 *Destination Blender data flow.*

The Triangle Engine works in a *deferred mode*, meaning that it executes commands from a buffer, rather than responding immediately to each of the commands that you issue. The normal flow of these commands usually looks similar to the list below:

1. Provide information about texture's format, location in memory, etc.
2. Load the texture.
3. Provide information about the texture's PIP (Pen Index Palette, or color table).
4. Load the PIP.
5. Set up the other hardware registers, such as the Destination Blend unit.
6. Provide vertices of one or more triangle strips or fans, to be rendered using the parameters described in Steps 1 - 5, above.
7. Issue a SYNC instruction. Refer to "Flow Control Instructions," later in this chapter, for more information about the SYNC instruction.
8. Return to Step 1.

The only step in the process that actually causes pixels to be rendered is Step 6. The other steps describe how those pixels will be rendered. Note that the SYNC instruction is needed after a collection of vertex instructions have been issued to the Triangle Engine, but it is described in more detail later.

Vertex Instructions

Since vertex instructions are what actually cause pixels to be rendered, we'll start by describing the CLT macros that create these instructions. For now, let's assume that the Texture Mapper, Destination Blender, and the other setup portions of the Triangle Engine are all set to allow these instructions to render correctly.

The general form of vertex instructions is one header word, followed by three or more vertices. The header word describes what the format of the vertices will be, a count of the vertices that follow, and also tells the Triangle Engine whether the vertices describe a triangle strip, or a triangle fan. Vertices contain x and y data, and may also contain RGBA color data, uv texture mapping data, and a $1/w$ perspective correction value (also used for depth buffering). The actual contents of a vertex instruction depends on the macro you choose to use.

The macro to create a vertex header instruction is of the form:

```
CLT_TRIANGLE (pp, stripfan, perspective, texture, shading, count)
```

`pp` – the address of a pointer to the current insertion point in a command list buffer. See the section on GState for more details about this command list buffer pointer. The address of the pointer is necessary, because CLT updates the pointer as it adds instructions to the command list buffer.

Note: *See Chapter 2 for more details about the command list buffer pointer.*

`stripfan` – an integer stating whether the vertices that follow should be interpreted as a triangle strip or a triangle fan. The constants `RC_STRIP` and `RC_FAN` should be used for this parameter.

`perspective` – a bit stating whether the vertices that follow contain a $1/w$ perspective correction value.

`texture`– a bit stating whether the vertices that follow contain U and V texture coordinates.

`shading` – a bit stating whether the vertices that follow contain R, G, B, and A (alpha) color data.

`count` – an integer count of how many vertices follow this header word. It is important to note that the Triangle Engine can work more efficiently on longer strips and fans than on shorter ones. If optimal rendering speed is desired, the average number of triangles per strip or fan should be 10 or more.

There are several forms of the CLT vertex macros. Each form covers one of the various triangle formats described by the header word. Some examples follow:

```
CLT_Vertex(pp, x, y)
CLT_VertexRgbaW(pp, x, y, r, g, b, a, w)
CLT_VertexRgbaUvW(pp, x, y, r, g, b, a, u, v, w)
```

`pp` – the address of a pointer to the current insertion point in a command list buffer. As with all CLT macros that take `pp`, it is updated as data is written to the buffer.

All of the remaining values for the `CLT_Vertex` instructions are represented as IEEE 754 standard 32-bit floating point numbers.

`x, y` – the x and y coordinates of the vertex of a triangle. They should be in the range of 0 to bitmap width, and 0 to bitmap height, respectively. Negative numbers are not allowed for `x` and `y`, so care has to be taken to clip triangles against the top and left edges of a bitmap. Triangles that extend beyond the right or bottom edges of a bitmap are clipped by the Triangle Engine, if they are within the range of 0.0 to 2048.0.

`r, g, b, a` – represent the red, green, blue, and alpha components for shading a triangle. This data might be different at each vertex, to allow for Gouraud shading of the triangles, or it can be the same at each vertex, in order to flat shade the triangles. The values should be in the range of 0.0 to 1.0 for each of these components.

`u, v` – represent the texture coordinates that should be applied to a triangle. The values should be in the range of 0 to texture width, and 0 to 1024, respectively. When perspective correction of the textures is desired, u and v should actually be sent to the Triangle Engine as u/w and v/w. Some textures can be tiled, meaning that its texel data repeats in the x and/or y directions. Note that if u and v exceed the width of the texture, the clamping mask is applied.

`1/w` – a perspective correction factor. As a result, the range for the `w` value is [0,1). When w = 0.0, a triangle is represented as being infinitely far away. As `w` approaches 1.0, the perspective correction applied to it becomes less and less, until eventually there is no correction done on the triangles.

Below is a fragment of code that adds two triangle strips to a command list buffer. Again, this code assumes that the texture mapper and destination blender have been correctly set up, and that the variable `listPtr` is a valid pointer to the current insertion point in a command list buffer.

```
{
  ...
  CLT_Triangle( &listPtr, RC_STRIP, 0, 1, 0, 6 );
  CLT_VertexRgba( &listPtr, 20.0, 20.0, 0.0, 0.0, 0.0, 1.0 );
  CLT_VertexRgba( &listPtr, 23.0, 60.0, 0.1, 0.1, 0.1, 1.0 );
  CLT_VertexRgba( &listPtr, 110.0, 10.0, 0.7, 0.7, 0.7, 1.0 );
  CLT_VertexRgba( &listPtr, 95.0, 80.0, 0.7, 0.7, 0.7, 1.0 );
  CLT_VertexRgba( &listPtr, 200.0, 20.0, 0.2, 0.2, 0.2, 1.0 );
  CLT_VertexRgba( &listPtr, 190.0, 60.0, 0.14,0.14,0.14,1.0);
  ...
}
```

This piece of code outputs a strip comprised of six untextured, shaded triangles. The triangles start at roughly 20 pixels into the bitmap from the left, and move right until they are about 220 pixels in from the left edge of the bitmap. As they near the center, they get brighter.

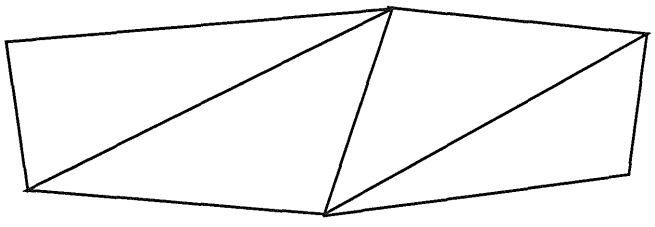


Figure 1-4 *Triangle strip*

State Control Instructions

The way in which a triangle is actually rendered in a bitmap is controlled by the state of the Triangle Engine. For example, the triangles in the previous sample code might have a texture blended with the shading, might be dithered, or might be rendered as partially translucent triangles. In this section, the methods for changing this state are described. The individual state-controlling registers are described in Chapters 3 and 4.

The state registers for the Triangle Engine are all 32-bits, although some features may only use some of the bits in a register, while others may be affected by data from more than one register. The Triangle Engine provides three ways of writing data to these registers, and so the CLT provides three methods as well:

```
CLT_WriteRegister( pp, reg, val )
CLT_SetRegister( pp, reg, val )
CLT_ClearRegister( pp, reg, val )
```

The first command, `CLT_WriteRegister`, stores the value `val` in the register `reg`. The second command, `CLT_SetRegister`, ORs the bits in `val` into the previous value of the register `reg`. The last command, `CLT_ClearRegister`, clears any bits in `reg` if the corresponding bits in `val` are set. For each of these calls, `pp` is the address of a pointer to the insertion point in a command list buffer.

These macros actually write a header word to the command list buffer, then the 32-bit quantity, `val`. This header word describes whether a Set, Clear, or Write should occur, at what address the change should occur, and how many consecutive registers are written. Each of the macros (`CLT_WriteRegister`, `CLT_SetRegister`, and `CLT_ClearRegister`) always write only one register at a time. If a user program wants to change the state of several registers at once, and these registers resided in consecutive addresses, a more efficient method would be to use:

```
CLT_WriteRegistersHeader( reg, cnt )
CLT_SetRegistersHeader( reg, cnt )
CLT_ClearRegistersHeader( reg, cnt )
```

Note that these CLT macros do not take `pp` as an argument, so user code has to look something like the example below:

```
{
  ...
  CLT_WriteRegister( pp, DBCONSTIN, 0xFFA01010 ); /* mostly red */
  *pp++ = CLT_WriteRegistersHeader( DBDITHERMATRIXA, 2 );
  *pp++ = 0xC0D12E3F;             /* DBDITHERMATRIXA was just set */
  *pp++ = 0xE1D0302F;             /* DBDITHERMATRIXB was just set */
  ...
}
```

Whenever these macros need to name a register, they can use one of the named register constants. In the CLT header file, all of the named registers are of the form `RA_<regName>`. For example, `RA_DBDITHERMATRIXA` is the register address of the top half of the destination blender's dither matrix.

Because some attributes might only use a few bits of one of the registers, the CLT header file also defines several bit shifts and masks that can be used to set just portions of a register. Each field within each register has been named in the form `FV_<regName>_<Attribute>`. For each of these field values, the shift and mask can be used to ensure that values do not write into other register attributes accidentally.

The following macros can be used in combination with the CLT_ClearRegister and CLT_SetRegister macros to only change a portion of register, so that the remaining fields within the register can remain intact.

```
CLT_Mask( reg, field )
CLT_Bits( reg, field, data )
```

The following code fragment sets only the red portion of the Destination Blender's BMultConstSSB0 register, to the value 0x40:

```
{
  ...
   CLT_ClearRegister( pp, DBBMULTCONSTSSB0,
   CLT_Mask(DBBMULTCONSTSSB0, RED ) );       /* Clear red channel */
   CLT_SetRegister( pp, DBBMULTCONSTSSB0,
   CLT_Bits(DBBMULTCONSTSSB0, RED, 0x40 ) ); /* OR in 0x40 for red */
  ...
}
```

For some of the field values, a constant makes more sense than a numeric value. These constants have also been defined in the CLT's header file, along with some macros to use them. The CLT_Const macro can be used to retrieve one of these constants to provide a value for the data field of the CLT_Bits macro. The following example illustrates one way to use the CLT_Const macro. It sets the InterFilter of the Texture Mapper to TriLinear filtering.

```
{
  ...
  CLT_ClearRegister( pp, TXTADDRCNTL,
    CLT_Mask( TXTADDRCNTL, INTERFILTER ) );
  CLT_SetRegister( pp, TXTADDRCNTL,
    CLT_Bits( TXTADDRCNTL, INTERFILTER,
      CLT_Const( TXTADDRCNTL, INTERFILTER, TRILINEAR ) ) );
  ...
}
```

For each of the state registers in the Texture Mapper and Destination Blender, the CLT header file also provides macros that take all of the necessary field values as arguments, and outputs the correct CLT_WriteRegister command. Using these simpler macros, in cases where all of the data in a register needs to be set at once, can make the code appear cleaner.

Flow Control Instructions

The Triangle Engine has some additional instructions that affect its data flow. These instructions, collectively known as the *flow control instructions,* start and stop the Triangle Engine, cause it to jump to a new place within a command list buffer, and can instruct it to actually load a PIP or texture from main RAM. The flow control instructions can use the same macros as the state control registers (described above) or any of the following macros:

```
CLT_Sync(pp)
CLT_Pause(pp)
CLT_Interrupt(pp, ival)
CLT_JumpRelative(pp, address)
CLT_JumpAbsolute(pp, address)
CLT_TxLoad(pp)
```

Caution: *It is important to understand the use of these registers, because using them incorrectly can cause the Triangle Engine to crash.*

The SYNC instruction tells the Triangle Engine to flush all of its internal pipelines. This command **must** be used to separate one or more consecutive triangle strips and/or triangle fans from any state control instructions that may follow the vertices. Failure to use the SYNC instruction, after vertices have been rendered, can lead to Triangle Engine crashes.

The PAUSE instruction works like the SYNC instruction, except that it also causes the Triangle Engine to stop at the word after the PAUSE. The TE remains stopped until it is restarted at some later time by the CPU. When a PAUSE is encountered, an interrupt can be generated if the Triangle Engine was set to cause an interrupt when it reached the end of a list. The command to tell the Triangle Engine to begin, once a PAUSE has been encountered, is a command that must be executed by the Triangle Engine Device Driver. Each command list buffer normally contains a PAUSE instruction after the last state control or vertex instruction in the buffer.

The INT (interrupt) instruction tells the Triangle Engine to cause an interrupt for the CPU. This should not be used by applications, since the Triangle Engine device driver currently does not handle special interrupts.

The JR (jump relative) instruction causes the Triangle Engine to jump to a different address within a command list to fetch instructions. The relative offset from the jump instruction (a 2's complement number), should first be loaded into the `RA_DCNTLDATA` register by using the `CLT_WriteRegister` macro. Note that this is normally not needed by applications.

The JA (jump absolute) instruction causes the Triangle Engine to jump to an absolute address to fetch further instructions. The address should first be loaded into the `RA_DCNTLDATA` register by using the `CLT_WriteRegister` macro. Note that this is normally not needed by applications.

The TLD (texture load) instruction is used throughout a command list buffer whenever it becomes necessary to load a new texture into the Triangle Engine's Texture RAM (TRAM). To load a texture successfully, it is normally necessary to set up several registers in the Texture Mapper first, such as the load address, etc. Once all of these registers are set up, you issue a TLD command before any vertex instructions attempt to use the new texture. Note that it is not necessary to SYNC immediately before or after a TLD instruction, unless it happens to be an instruction that follows a vertex instruction.

Command List Snippets

So far, this document has described the CLT macros as tools for writing into a command list buffer. There are many cases, however, where it is useful to build a block of Triangle Engine commands once, then re-use the pre-built data several times. One way to prevent your code from having to re-compute these macros, each time a repeated set of commands needs to be applied, is to write the commands to a *Command List Snippet*, or `CltSnippet` data structure. These snippets can be dynamically allocated and filled once with commands. Then, only a simple `memcpy` is necessary to place the data into a command list buffer. In the M2 Graphics Pipeline code, several command list snippets are saved off and re-used, to help save computing cycles. For example, one command list snippet is provided that disables texturing. Whenever a non-textured object follows a textured object, one simple call copies data into the command list buffer to disable the Texture Mapper. A `CltSnippet` is defined as follows:

```
typedef struct {
  uint32*  data;
  uint16    size;        /* Number of words used */
  uint16   allocated;   /* Number of words allocated */
} CltSnippet;
```

To allocate a new command list snippet, call:

```
int CLT_AllocSnippet(CltSnippet *s, uint32 numWords)
```

This routine allocates enough memory for the specified number of commands to be placed into this `CltSnippet`.

There may be cases when it is useful to create one `CltSnippet`, then re-use variations of it several times. To copy one snippet into another, call:

```
void CLT_CopySnippet(CltSnippet *dest, CltSnippet *src)
```

The `dest CltSnippet` must already have been allocated, and its data area must be at least as large as the `src CltSnippet`.

Once these snippets are built, the routine that copies a command list from the `CltSnippet` into a command list buffer is:

```
void CLT_CopySnippetData(uint32 **dest, CltSnippet *src)
```

No check is done to ensure that there is enough space remaining in the command list buffer for the whole snippet, so it is the responsibility of the application to do this before attempting to copy the data into the command list buffer.

Once a command list snippet is no longer needed, the memory allocated to it can be freed with a call to:

```
void CLT_FreeSnippet(CltSnippet *s)
```

Chapter 2

Graphics State (GState)

The Graphics State (GState) object helps users manage the command list buffers that store the data used by the Triangle Engine to render graphics.

The main job of the GState is to provide an easy way of creating command list buffers, filling them with Triangle Engine instructions, and eventually sending them to the Triangle Engine device driver, so that the instructions can be executed. The GState is designed to give the user flexibility in how memory is used for command list buffers while keeping the overhead, both in memory and in CPU cycles, to a minimum.

This chapter describes how to use the GState to create command list buffers, and how to correctly use them so that user code, the 2D and 3D Pipelines and Frameworks, and the Font Folio, can all work together to render graphics through the Triangle Engine.

This chapter contains the following topics:

Introduction

The 2D Pipeline and Framework, the 3D Pipeline and Framework, the Command List Toolkit, and the Font Folio, all write commands to the command list buffers. Providing one standard way for all of these components to write to and send command list buffers results in the following benefits:

- Commands from all of the different sources can be intermingled throughout the process of rendering a scene
- Memory used for command list buffers can be shared
- Code used to manage command list buffers can be shared

The GState is a data structure and collection of routines that encapsulate command list buffers in a way that all of these libraries and folios can understand and use. Routines are provided to create command list buffers, determine where the current insertion point is in a command list buffer, and to send a command list buffer to the Triangle Engine device driver. For optimal performance, command lists should be double-buffered, so that while the Triangle Engine is executing the instructions in one buffer, the CPU can begin filling the second buffer with the next batch of commands. The GState supports double-buffered command lists.

A GState object looks like the following structure:

```
typedef struct GState {
  Err       (*gs_SendList)(struct GState*);
  CmdListP  gs_EndList;
  CmdListP  gs_ListPtr;
} GState;
```

In addition, there are several private fields which must be accessed through the interface routines provided.

Creating and Initializing a GState

Before command lists can be created, a GState needs to be created. To create a GState, an application should call:

```
GState* GS_Create(void);
```

Memory is allocated for a GState structure, and the structure is initialized to a default state. Users should *not* allocate a GState as a stack variable, because the structure contains many fields that are private. The next step is to allocate one or more command lists on a GState. The following routine allocates the specified number of lists, and assigns them to a GState:

```
Err GS_AllocLists(GState* g, uint32 numLists, uint32 listSize);
```

Note that `listSize` is specified in WORDS, not bytes. Command lists are just blocks of user memory, like any other blocks allocated with `AllocMem()` or `malloc()`.

The other setup necessary for a GState is to specify what bitmap the command lists should be rendered into, and optionally, what bitmap to use as a Z-buffer. These buffers are set and retrieved with the following calls:

```
Err GS_SetDestBuffer(GState* gs, Bitmap* bm);
Err GS_SetZBuffer(GState* gs, Bitmap* bm);
Bitmap* GS_GetDestBuffer(GState* gs);
Bitmap* GS_GetZBuffer(GState* gs);
```

When double-buffering frame buffer bitmaps, it is usually desirable to create *tear-free rendering*. To accomplish this, an application can associate a signal bit with a GState and the Graphics Folio. When the Graphics Folio finishes displaying a bitmap, it sends a signal to a task, stating that the previously-displayed bitmap is completely off the screen, and thus safe to render into. If a signal is allocated for use as a display signal in the Graphics Folio, it can be associated with a GState by calling:

```
Err GS_SetVidSignal(GState* gs);
```

If the signal is set on a GState, the GState waits before sending the first command list buffer for a bitmap.

Using a GState

Once a GState has been set up, data can be written to it's command list buffers by writing 32-bit words to a pointer to the current insertion point in the current command list buffer. The GState maintains such a pointer in the field `gs_ListPtr`. Most of the Command List Toolkit macros, that write data to a command list buffer, take the address of this field as one parameter, and update the `gs_ListPtr` as they write data to the buffer.

The 2D and 3D Pipelines already do their own checks to ensure that there is enough space in a command list buffer before issuing a block of commands. However, when using CLT, it is up to the application to ensure that there is enough room remaining in the buffer. The following call ensures that there is enough memory for the requested number of words of space:

```
void GS_Reserve(GState* gs, uint32 numWords);
```

If sufficient space isn't available, the current command list buffer is sent to the Triangle Engine, and the next available command list is made current. The `gs_ListPtr` field then points to the beginning of this new command list buffer. Note that `GS_Reserve()` will fail if an application requests more words than are actually available in an entire list. If this happens, the results are unpredictable. It is a good practice, while developing an application, to allocate plenty of space to the command list buffers. You can reduce the size of the buffers once the rest of the code has stabilized. At this point, you can make a good estimate of what the minimum buffer size needs to be. Note that `GS_Reserve()` does not actually advance the insertion point by any amount of space. It is up to user code, or CLT macros, to advance the insertion pointer as data is written to the buffer.

When `GS_Reserve()` does not find enough space available in the current command list buffer, it calls the function pointed to by the `gs_SendList` field. This field normally points to the following routine:

```
Err GS_SendList(GState* gs);
```

This routine can be called at any time by the application. It flushes the current command list buffer to the Triangle Engine. `GS_SendList()` sends a command list buffer to the Triangle Engine to be rendered. This routine can optionally wait for a video signal saying that it is safe to start rendering to a buffer, and will ensure that `gs_ListPtr` is pointing to an available command list buffer before

returning. GS_SendList() calls SendIO to send the command list buffer to the Triangle Engine device driver. The following routine can be used to wait for all current and pending IORequests to complete:

```
Err GS_WaitIO(GState* gs);
```

Normally, this routine is called before attempting to display a buffer in a double-buffering scheme, so that the application can ensure that Triangle Engine rendering is not visible to the end user.

If it becomes necessary to write your own routine to send a list to the Triangle Engine, the following few routines might be helpful:

```
CmdListP GS_GetCurListStart(GState* g);
```

This routine returns a pointer to the beginning of the current command list buffer. In order to send a list to the Triangle Engine device driver, it is necessary to specify the start of a command list buffer, the end of the buffer, and some frame buffer information. Look at the source for GS_SendList() for exact details on how to send a command list.

```
void GS_SetList(GState* g, uint32 idx);
```

GS_SetList() sets which command list to use for the given GState. The idx field is in the range of 0...(num cmd lists - 1).

```
uint32 GS_GetCmdListIndex(GState* g);
```

This routine returns the index of the command list currently in use. Normally, you would call GS_SetList() with (this index + 1) % (num cmd lists) after sending the current list.

Synchronizing the Triangle Engine to Video Signals through GState

As mentioned earlier, it is possible to set up a GState to provide tear-free double-buffering of frame buffers. The basic principle is: Before a bitmap can be used as an output buffer, the GState ensures that it is not currently being displayed on the screen. The Graphics Folio can send signals that, if used correctly, allow the GState to always write only to buffers that are not up on the display. In combination with the correct calls to the Display Folio, buffers are swapped onto the screen with no visible tearing. The basic steps to follow to achieve this tear-free double-buffering are:

- Allocate a signal bit by calling AllocSignal().
- When a view is created, define this signal as the display signal. This is done by specifying the signal as the argument for the VIEWTAG_DISPLAYSIGNAL tag when calling CreateItem for the view.
- Create a GState by calling GS_Create().
- Associate the signal with the GState. This is done by calling the routine:

  ```
  Err GS_SetVidSignal(GState* gs, int32 signal);
  ```

- At the beginning of each frame to be displayed as a frame buffer, call:

  ```
  Err GS_BeginFrame(GState* gs);
  ```

If rendering to a buffer that is not going to be immediately displayed, do not call GS_BeginFrame(). When GS_SendList() sees that GS_BeginFrame() has been called, and a signal is associated with a GState, then GS_SendList() waits

until that signal is activated by a call to `ModifyGraphicsItem()`. If a bitmap is going to be rendered into, and `ModifyGraphicsItem()` is not called to display it, the signal is never sent to the GState, and the application effectively hangs.

Cleanup Routines

To free the memory allocated to the command list buffers for a GState, call:

```
Err GS_FreeBuffers(GState* gs);
```

If, for some reason, it is necessary to re-allocate new command list buffers, GS_AllocLists() can be called again.

Once an application is completely finished using a GState, the GState can be freed by calling:

```
Err GS_Delete(GState* gs);
```

If `GS_FreeBuffers()` has not been called prior to `GS_Delete()`, it is called implicitly as the GState is freed in memory.

Combining CLT, the 2D and 3D Pipelines and Frameworks, and the Font Folio

The 2D and 3D Graphics Pipelines, and the Font Folio, are all built on top of GState. One benefit of this is that calls from these various sources can be intermingled throughout a frame. For example, if an application is using the 3D Pipeline to render some scenery for a flight simulator, and then going to use the Font Folio to display a status bar with altitude, air speed, etc., the application could set up the scene, and tell the 3D Pipeline to render it. Before the command lists are flushed, the application could then call the Font Folio routines to display the appropriate text. A final call to `GS_SendList()` would then render all of the commands from the Pipeline and the Font Folio into the bitmap at once.

Chapter 3

Textures

In M2, a *texture* is a two-dimensional image that is used to modify the color or the transparency of a 3D object. A texture follows the surface that it is mapped to—often folding, bending, or wrapping around to follow the contours of a 3D object.

This chapter contains the following topics:

Introduction

A texture can be something as simple as a color (with or without various degrees of shading) or something as complex as (or more complex than) a tiled pattern, a newspaper page, a brick wall, or even a previously-rendered frame buffer. The texture pipeline is capable of processing from 1 pixel every 3 ticks to 2 pixels per tick, depending on the filtering being applied.

In the M2 system, textures are stored in arrays, and each element of the array in which a texture is stored is called a *texel*. Each texel can be up to 32 bits long. For more information about the structure of a texel, see "How Textures Are Stored in Memory."

The coordinate system for a texture is shown in Figure 3-1. By convention, the horizontal and vertical coordinates of a texture are expressed as *U* and *V*, as shown in the diagram. If *U* is a variable that is used to represent horizontal coordinates and *V* is a variable that used to represent vertical coordinates, the point where<*(U, V*> = <*0, 0*> is at the top-left hand corner of the texture's mipmap, and the address of that point is considered the base address of the texture. This means that in the case of an 8-bit texture, the map is stored in memory with *U* increasing by one from one memory location to the next.

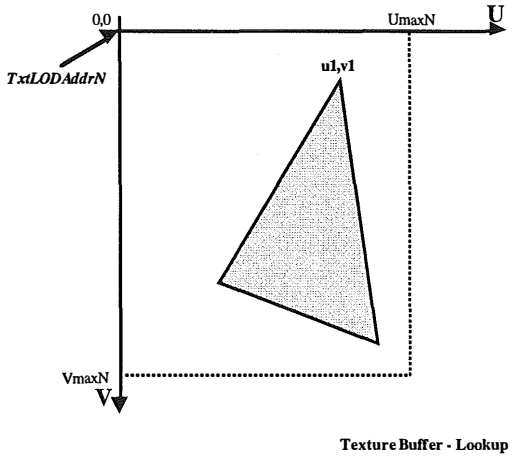


Figure 3-1 *Texture Coordinates*

Note: *U and V coordinates must be in the range from 0 to 1023.*

For more information about how M2 stores textures in memory, see "How Textures Are Stored in Memory." See Appendix A for the register setup used to turn texturing on and off.

Texture Mapping

Texture mapping is the process of mapping a texture onto a 3D object displayed on a screen.Texture mapping can be tricky because a texture is a 2D rectangular region that must often be mapped to a 3D non-planar surface with various kinds of irregular features. For example, Figure 3-2 shows how a texture might be mapped to a pyramid-shaped 3D object.

The texture shown in the rectangle on the left has been mapped to the pyramid shown on the right. Notice that the texture obtained from the texture on left is wrapped around to match the shape of the pyramid and is distorted when it is mapped to the pyramid.

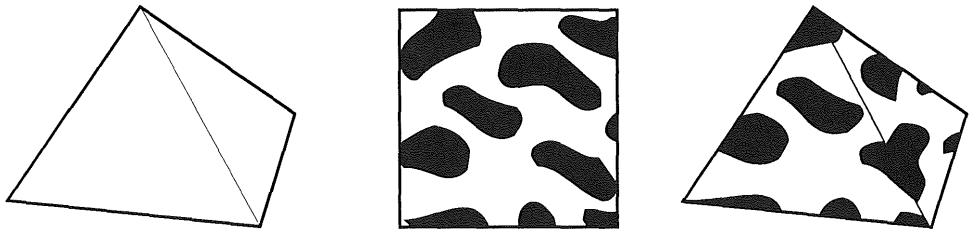


Figure 3-2 *The texture-mapping process*

Using Filtering in Texture-Mapping Operations

Depending on several factors—including the size of the texture being used, the 3D object's distortion, and the size of the screen image—some texels in a texture might be mapped to more than one pixel in the object's image, and some parts of the object might be covered by multiple texels.

Because a texture is made up of discrete texels, the M2 Triangle Engine performs various kinds of filtering operations to map texels to pixels in images of 3D objects. For example, if many texels correspond to a single fragment of an object, they are averaged down to fit the number of pixels into which they are mapped. If texel boundaries fall across fragment boundaries, things get even more complicated, and a weight average of the affected pixels is performed.

Replicating Textures

M2 allows you to repeat a texture over the surface of an object. Replicating a texture in this way is called *tiling*. If a texture is to be tiled, it must have a size that is a power of two in both directions. If this condition is met, a simple mask is used on the texture's final U, V coordinate (refer to Figure 3-1) before the address calculation is performed to mask out some most signifcant bits (MSBs).

Mipmapping

A textured object, like any other object, can be viewed at different distances from the camera. If a textured object moves away from the viewer, or if the viewer moves away from the object, the number of pixels covered by a single texel decreases. To make this change in texture mapping correlate properly with the reduction of the object, M2 filters the texture map down to an appropriate size as the object grows smaller, without introducing any visually disturbing artifacts.

To prevent the generation of such artifacts, M2 uses prefiltered texture maps called *mipmaps* (see Figure 3-3).

Mipmaps (MIP stands for the Latin *multim im parvo*, meaning "many things in a small place") are maps that provide multiple sizes of the same texture in powers of 2. The size of a mipmap can range from 1 x 1 texel to the size of your highest-resolution map. For example, a set of four mipmaps could have sizes of 64 x 16, 16 x 4, 8 x 2, and 4 x 1.

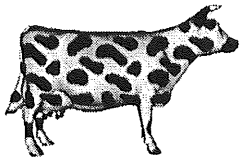
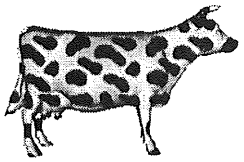
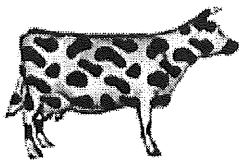
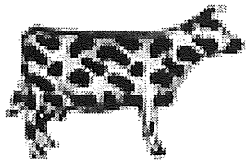
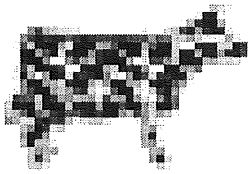
Mipmap Level	Resolution	Actual Size	Magnified
0	Full		 *Not Magnified*
1	1/2		 *Magnified by 2*
2	1/4		 *Magnified by 4*
3	1/8		 *Magnified by 8*

Figure 3-3 *Mipmaps*

Figure 3-3 shows how mipmaps provide multiple sizes of the same texture image. Notice that each mipmap is one-fourth the size of the next higher-level mipmap: one-half its height, and one-half its width.

Levels of Detail (LODs)

Each version of a texture's image is called is called a *level of detail* (LOD). The original version of the texture, which has the finest detail, is called LOD 0. Other LODs are numbered upward consecutively, as shown in Figure 3-3.

M2 hardware supports the use of up to four levels of detail (LODs) at a time, but you can control which LODs are used by specifying the index of the finest LOD and the number of LODs to use.

Using Mipmaps

When you have created a set of mipmaps, here's how M2 uses them: Starting with the highest-resolution texture, the desired number of filtered and minified versions is created. Each version is reduced from the size of the previous version by a factor of two in both the *U* and *V* directions (*U* and *V* are the texture-coordinate equivalents of *x* and *y* in the world coordinate system).

When an object is texture-mapped and a set of mipmaps is available, M2 can automatically determine which mipmaps to use, on a per-pixel basis, based on the size (in pixels) of the object being mapped. When the image of the object gets small enough, M2 can switch to the next smallest mipmap to map the object's texture. When the object grows large enough, M2 can switch to the next-largest mipmap.

Mipmap Filtering

Mipmapping improves the appearance of rendered images considerably. But when you map a texel to a surface, there is rarely an exact match between the (*U*, *V*) coordinates of the texel being mapped and the (x, y) coordinates of the mipmap being used in the mapping operation. In other words, the precise texture-to-map ratios shown in Figure 3-3 are ideal ratios that actually seldom occur.

When that is the case—as it usually is—an operation known as *mipmap filtering* is often required to avoid aliasing.

Although some mipmap filter is needed in most texture-mapping operations, the amount of filtering you should perform in any particular situation can vary, depending on your needs. Also, M2 supports four different filtering modes, each designed for use in a specific kind of situation.

The four filtering modes offered by M2 are:

- *Nearest (point) filtering*, in which each texel in a texture is mapped to a surface using the best mipmap that is available, ignoring the possibility that the match my not be precise.
- *Linear filtering*, which uses two mipmaps—one that is actually too large and another that is actually too small—to average the color of each texel being matched
- *Bi-linear filtering*, which uses a similar technique to the one used in linear filtering, but applies a weighted average of the colors of the texels surrounding the texel being mapped in order to obtain a color value for that pixel that is more precise
- *Quasi tri-linear filtering*, which uses a slower but more precise algorithm to obtain an even more precise weighted value for the texel being mapped.

Each of these filtering modes has advantages and disadvantages. As you move from the fastest filtering mode (nearest [point] filtering) to the slowest, the quality of the rendered image improves, but the time required to filter the data increases. Nearest (point) filtering is the fastest mode, but offers the lowest resolution. Quasi tri-linear filtering is the slowest method, but offers the highest resolution.

Table 3-1 lists the four texture-filtering modes used in M2 and compares their speed and characteristics. Note that the bi-linear and quasi tri-linear filtering modes require a 2-by-2 array of texels each.

Table 3-1 *Texture Filtering Modes*

Type	Speed	Texels Required	Operation
Point	2 pix/tick	*T[U,V, n]*	None Out = *T[U,V, n]*
Linear	1 pix/tick	*T[U, V, n],* *T[U/2, V/2, n+1]*	When a sample lies between two mipmaps, the distance of the sample from each map is used to compute a linear blend between the two maps.
Bi-Linear	0.5 pix/tick	*T[U, V, n],* *T[U+1,V, n],* *T[U,V+1, n],* *T[U+1,V+1, n]*	The fractional placement within a map is used to blend four adjacent texels from one map together.
Quasi Tri-Linear	0.33 pix/tick	*T[U, V, n],* *T[U+1,V, n],* *T[U,V+1, n],* *T[U+1,V+1, n],* *T[U/2, V/2, n+1]*	A mixture between the two above modes. A bi-linear blend is performed at LOD*n*, then a linear blend is performed with a texel from-LOD*n*+1.

Note: *For texels at the edge of a texture, all the information required may not be present. You may be able to ignore this anomaly, or you may want to add a one-pixel guard region around the whole texture. By adding a guard region, you can guarantee that all required information is present. (A guard region is actually required for texture tiling; for details, see "Runtime Carving" on page 38.) In the M2 hardware, the address clamps in both the U and V directions, so if you do not add a guard region, replicating the last texel is the default mode.*

How M2 Mipmap Filtering Works

Figure 3-4 shows the various kinds of filtering operations that are used for mipmapping in M2. The diagram is a stylized representation of a perspective view that extends from the camera out into the distance. Such a triangle could exist, say, as part of the floor of a large building.

The problem that such a triangle generates for the M2 hardware is that at some points the cameraa is not between two LODs. At those points, linear interpolation between maps is is not possible. (It is assumed here for sake of argument that Region 3 in the diagram is the desired operating region. At that point, the camera lies between maps of different LODs, so you can perform linear or tri-linear interpolation).

If the sampling process takes you into Region 2, that means you are trying to magnify beyond the finest level of detail, which is not possible. That is the point where interpolation no longer works, and where point sampling the texture would lead to blockiness. At that point, as in the previous illustration, the best you can do is to switch to bi-linear interpolation in such a way that the fraction bits of the *(U, V)* coordinates are used to position a box filter over four neighboring texels.

Note that this alternative does not create any more information; it simply blurs the information. But that is preferable to the blockiness that would result from point sampling.

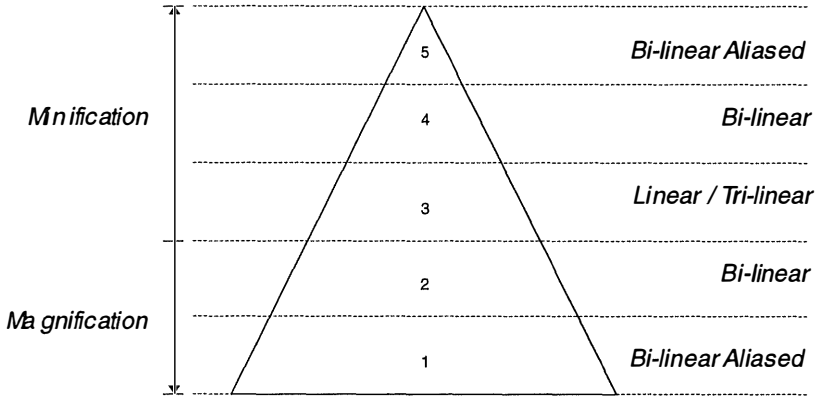


Figure 3-4 *Tradeoffs in filtering operations*

Monotonic Region Variations

In Figure 3-4, notice that that within any one span, the region variation is *monotonic*—that is to say, once you are out of one region, no more pixels within the horizontal span you have entered can fall within that region again. It is possible, however, to jump past several regions at once. For example, you can go from Region 1 to Region 5 in one pixel step. However, you can't go from Region 3 to Region 4 and back to Region 3 again within a span.

Figure 3-4 also illustrates some problems that can arise in using mipmap filtering. Assume that the scene shown in the diagram has such a long perspective that the texture you are applying to the nearest rectangle is magnified beyond the level of detail provided by the finest mipmap available. At the same time, assume that the texture you are applying to the farthest rectangle is minified beyond the level of detail provided by the coarsest mipmap available.

Problems in Minification

For minification, a problem arises if the programmer has not provided all the LODs that are needed to bring resolution down to the 1-by-1 map. Again, the M2 hardware can detect the fact that it has fallen out of the range in which linear interpolation is possible. When that hapens, the hardware falls into Region 4, where it must now either point-sample the coarsest available LOD map (an operation results in aliasing) or (once again) use bi-linear interpolation.

Because the hardware can easily detect any of the conditions that have just been described, it is possible for the hardware to select the best possible algorithm for each region. But blindly trusting the hardware to make such a selection has drawbacks that arise from the performance differences between the various interpolation schemes.

If you don't want to trust everything to the hardware, you can specify (by setting a register) exactly which algorithm to select for each region. Table 3-2 shows all the legal selections you can make. Note that regions 1 and 5 are really part of 2 and 4 respectively.

Table 3-2 *Registers for specifying LOD Regions*

Region	Point	Linear	Bi-Linear	Q Tri-Linear
1	•		*	
2	•		*	
3	•	•	*	•
4	•		*	
5	•		*	

Warning: *For a filtering operation that involves perspectives such as those shown in Figure 3-4 on page 23, you will probably make the boundaries between regions visible if you change filter modes. When you are working with a static image, this side effect may not be very noticeable, but in a moving image the effect can be serious more serious because each region will have different motion artifacts.*

Using Mipmap Filtering

Now that we have seen how mipmap filtering works, we are ready to examine each of the varieties of filtering available in M2: nearest (point) filtering, linear filtering, bi-linear filtering, and quasi tri-linear filtering. This section describes each of these kinds of filtering in more detail.

Nearest (Point) Filtering

Nearest (point) filtering is the simplest kind of filtering operation. You can always perform nearest (point) filtering.

In a nearest (point) filtering operation, the filtered texture values map directly to the color and alpha values of the texel containing the point. This point is represented by the coordinates (U, V) in the equation below.

$$LOD = [U, V, n]$$

U and V are the texel coordinates and n is the level of detail (LOD) of the mipmap being used for the mapping operation.

Linear Filtering

When a filtering operation is performed, the (U, V) coordinates for adjacent destination pixels may be such that the computed level of detail falls between two LODs. For example, assume that i is the distance of the incrementation between two horizontal texels being mapped to a surface. If $(U(i+1) - U(i))$ is 3, the computed level of detail is in between 1 and 2. In such a case, linear filtering can be used to blend the texel data between the two bounding LODs. Linear filtering is illustrated below.

$$T = T[U, V, n] \bullet (1 - blend) + T[U, V, n + 1] \bullet blend$$

n is the level of detail (LOD) of the mipmap being used for the mapping operation, just as it was in the previous equation.

Linear filtering works by taking samples of the texture at (*U*, *V*, *n*) and at (*U*, *V*, *n* + 1). The sample values are combined in the ratio provided by the *blend* variable, which is computed based on the distance of the computed LOD from the finer LOD, in accordance with the preceding equation.

Requirements for Linear Filtering

Because linear filtering interpolates between two levels of detail, it cannot always be used for all pixels in a rendered triangle. In order to use linear filtering, the following requirements must be met:

- The texture being applied must have more than one LOD.
- The pixels being rendered must scale the texture such that it is larger than the finest LOD, and smaller than the coarsest LOD.

An Example of Linear Filtering

As an example of linear filtering, assume that a triangle is being rendered using a texture map that contains two levels of detail. Also assume that the dimensions of LOD 0 are 20 x 20 texels. That means that the dimensions of LOD 1 are 10 x 10, because each subsequent LOD must be half the width and half the height of the previous LOD. So the triangle to be rendered always uses texture coordinates that range from (0,0) to (19,19).

Note: *For the sake of simplicity, this example ignores the effects of the perspective correction factor, W, and assumes that an increase of 1.0 in triangle coordinate space maps directly to an increase of 1.0 in screen coordinates.*

Now assume that this triangle is to be rendered in such a way that its screen dimensions will be larger than 20 pixels wide or 20 pixels high. If that is the case, the largest texture LOD will have to be scaled *up* to fit the triangle. In M2 terminology, we say that the *magnification filter* is used.

In contrast, if the triangle is rendered in such a way that its screen dimensions are smaller than 10 pixels wide or 10 pixels high, the smallest LOD must be scaled *down* to fit the triangle. In M2 terminology, we say that the *minification* filter is used.

Now let's consider a third case. If the triangle we are discussing is rendered in such a way that its screen dimensions are between 10 and 20 pixels in the horizontal and vertical directions, the M2 *interfilter* is used. Note that when the Triangle Engine determines which LOD is needed, horizontal LOD and vertical LOD are computed independently, so it is possible that a pixel gets magnified horizontally, and minified vertically.

There is only one case in which linear filtering can occur (or, to put it into M2 terminology, there only one case in which the *linear filter* is used). That case arises when there is a texture map larger than the rendered triangle's needs, or when there is a texture map smaller than the rendered triangle's needs. When either of those texture maps is needed, the M2 linear filter can interpolate between the two. When a texture map needs to be magnified and there is no larger texture map available to interpolate the finest LOD, or when a texture map needs to be minified and there is no smaller texture with which to interpolate the coarsest

LOD, the linear filter can be used. Because certain filter types only make sense when a texel is between LODs (in other words, when the interfilter is used), each of the three filter regions can independently select one of the available filter modes.

Note: *Linear filtering takes twice as long as nearest (point) filtering.*

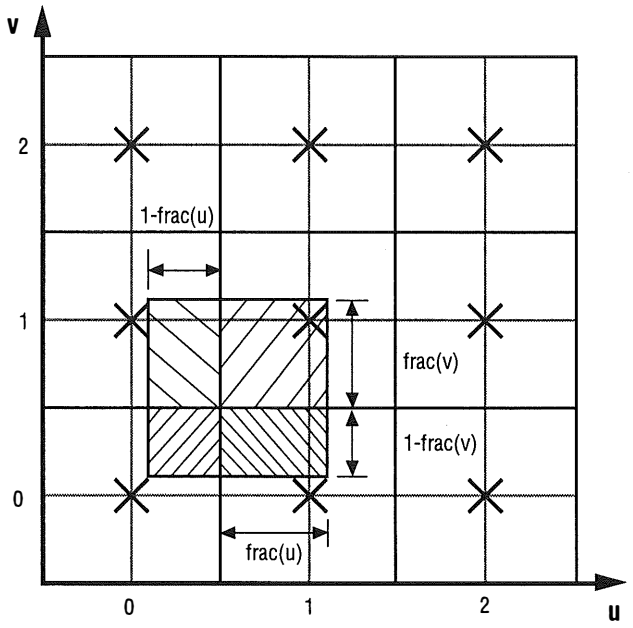


Figure 3-5 *Bi-linear filtering.*

Bi-linear Filtering

Bi-linear filtering (Figure 3-5), like nearest (point) filtering, is a type of filtering that can always be performed. Bi-linear filtering is equivalent to a one-pixel area filter that takes the weighted average from four adjacent texels, as shown below.

$$Tout = T[U, V, n] * (1 - fu) * (1 - fv) + T[U, V + 1, n] * (1 - fu) * fv + T[U + 1, V, n] * fu * (1 - fv) + T[U + 1, V + 1, n] * fu * fv$$

In a bi-linear filtering operation, the fractional parts of (*U, V*) are used to position the center of the filter, as shown in Figure 3-5. Note that the texel center is at *(.0,.0)*. Given the fractional parts of *U* and *V*, the four shaded areas can be constructed. The output value is then the sum of all four fractional areas multiplied by the corresponding pixel values.

Note: *Bi-linear filtering is four times slower than nearest (point) filtering. For a precise comparison of the speeds of different filtering modes, see Table 3-1.*

Quasi Tri-linear Filtering

Quasi tri-linear filtering is a cross between linear filtering and bi-linear filtering. In a quasi tri-linear filtering operation, a bi-linear blend is performed at the closest level of detail (determined by looking at the blend value) and point sample a texel

at the further LOD. A linear blend is then performed on the two results. This procedure results in the highest-quality resampling of the texture, but is somewhat slower than a straight bi-linear blend (see Table 3-1).

Quasi tri-linear filtering provides the best possible texturing quality, but it cannot always be used for all pixels in a rendered triangle. Because quasi tri-linear filtering performs a linear blend between two LODs, it can only be specified as the interfilter.

Perspective Correction

The M2 Triangle Engine has two modes of operation. The default mode is called *Perspective On Mode*. In this mode, M2 calculates the texture locations with perspective correction. The second mode is called *Perspective Off Mode*.

Perspective On mode provides perspective-correct *U* and *V* coordinates that are appropriate for mapping a texture onto a 3D surface. In Perspective Off mode, the texture values that are passed to the Texture Mapper are the raw *U/w* and *V/w* values, before division by *1/w*. Perspective Off mode can be useful in a 2D environment when you want to use a texture as a sprite and map it directly to the screen.

See Appendix A for the register setup used to turn perspective correction on and off.

Computing the *U* and *V* Coordinates for Texture Mapping

InPerspective On mode, the *U* and *V* coordinates are typically computed as

(U/w) / (1/w)

and as

(V/w) / (1/w)

This computation results in perspective-correct *U* and *V* coordinates that are appropriate for mapping a texture onto a 3D surface. When the Pespective Off mode is selected, the *1/w* value is not used, and the *U/w* and *V/w* coordinates are passed directly into *U* and *V*.

Using Perspective Off Mode in 2D Operations

Perspective Off mode can be useful in 2D rendering when the user wants to step through a flat texture and map it directly to the screen. It can also be useful in 3D mode at extremely large distances.

You can use Perspective Off mode when you want to generate successive integer *U* and *V* values for each succeeding pixel of a span.

The easiest way to accomplish this is goal is to use Perspective Off mode, but to set the *U/w* and *V/w* values at the vertices of the triangles in such a way that the *d/dx* values of these parameters are the same value: namely, 1.

You can accomplish this same effect in Perspective On mode by setting the *1/w* value as close to 1 as possible, making *(U/w) / (1/w)* is basically equal to *U/w*. However, this technique precludes the use of the *1/w* value for z-buffering in this situation. If z-buffering is not needed, using Perspective Off mode can eliminate the need for passing any *1/w* information into the Triangle Engine at all.

How Textures Are Stored in Memory

The M2 hardware supports texture sizes of up to 1,024 by 1,024 texels and up to four levels of detail (LOD). All the LODs that you use simultaneously must fit into a 16k block of dedicated RAM within the Triangle Engine called texture RAM, or TRAM.

In M2, the TRAM must be demand-loaded by software. Each successively coarser LOD must be exactly half the size in *U* and *V* from the previous level.

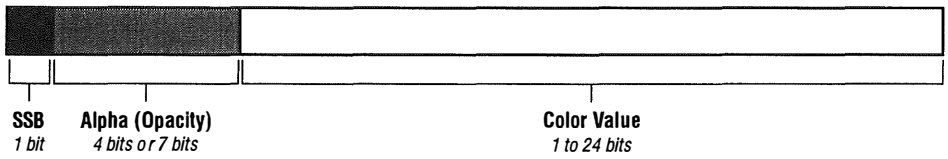
The programmer specifies the size of the coarsest LOD and the number of LODs to load. The size for the finer LODs are calculated automatically. Any aspect ratio texture may be specified.

When you load a texture into TRAM, you must load all adjacent LODs. For example, you cannot load just LOD0 and LOD2—you must also load LOD1. It is important to comply with this requirement; if you do not, the hardware produces unpredictable results.

When a texture is loaded, the *Umax* and *Vmax* register fields (see TXTUVMAX - Texture Loader Width Register (0x0004_642C)) are loaded with the dimensions of the coarsest map that is loaded. The finer mipmap sizes are given by multiplying by 2**(*LODmax* - *LODN*) where *LODN* is the current level of detail. Each texel in a texture can contain from one to three components. If a texel has three components, they are:

- A *source selection bit* (SSB) that is generally used to select between two sets of constant registers (for more information, see "Texture Mapping"). An SSB, when used, is always 1 bit long.
- An *alpha component* that specifies the texture's opacity. An alpha component can be either 4 bit long or 7 bits long.
- A *color component* that has two possible uses. It can specify the colors used in the texture or can be treated as an index into a palette index table (see "PIP Tables"). A color component can be 1 to 24 bits long.

The length of a texel's alpha component and color component depend on two things: whether the texel is compressed or uncompressed, and what kind of information the color value of the texel contains. illustrates the three parts of a texel.



The three possible components of a texel

Texture Formats

Textures in memory may be of two types: compressed or uncompressed. Texels within a texture may be of two types: literal or indexed. The term *texture formats* encompasses all four of these texture types. Table 3-3 lists and describes all four texture formats used in M2.

Table 3-3 *Texel Types*

Type	Description
Literal	Texels are stored as real color values. Only two formats are supported: five bits per color component or eight bits per color component.
Indexed	Texels may be stored using a arbitrary number of bits per texel up to eight bits. These get expanded to 8 bits per color component by using the PIP.
Compressed	Texels are stored using a run length encoding. Multiple bits per texel (types) may be used within the texture. These are expanded to the largest texel size used when the texture is read into the internal memory. Literal formats can not be used in compressed textures.
Uncompressed	Texels are stored in one format only. The number of texels is exactly the height * width.

In addition to color information, each texel format shown in Table 3-3 may also have two more pieces of information associated with it: alpha (which may be four or seven bits), and a Source Select Bit (SSB), which is a one-bit value that has multiple uses (in general, the SSB is used to select between two sets of constant registers for each stage defined for each register; for more details, see "How PIP Tables Work."

Table 3-4 lists all the combinations of color, alpha and SSB values that a texel can have.

Table 3-4 *Color, Alpha, and SSB Values that Can Be Assigned to a Texel*

Color	0	1	1	2	0	3	4	6	5	1	2	0	3	4	7	8	4	7	8	8	15	24	24
Alpha	-	-	-	-	4	-	-	-	-	4	4	7	4	4	-	-	7	4	4	7	-	-	7
SSB	1	-	1	-	-	1	-	-	1	1	-	1	1	-	1	-	1	1	-	1	1	-	1
Total	1	1	2	2	4	4	4	6	6	6	6	8	8	8	8	8	12	12	12	16	16	24	32

How Texels Are Stored in Memory

Uncompressed texels and compressed texels are stored in memory in different ways. Uncompressed texels are stored as arrays of texel data, but compressed texels are stored as texels of different types that are run length encoded. This section describes both these methods of storing texels in memory.

How Uncompressed Texels are Stored

Uncompressed textures are stored in memory as arrays of texel data. The number of bits per texel may vary from 1 to 32. Figure 3-6 shows how uncompressed texels are stored in memory.

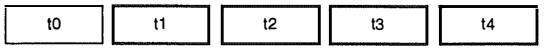


Figure 3-6 *Storage format of an uncompressed texel*

How Compressed Texels are Stored

Compressed textures consist of texels of different types (or formats) of texels that are run-length-encoded. This storage format allows the source texture to contain texels of different depths, so that portions of a texture that need more detail can use higher texel depths.

When texels are stored in memory, each sequence of texels that have the same type and value is called a *run*. Each new run is preceded by a control byte that specifies the run length and the type. Portions can be encoded to use less memory by switching to a different texel format on a per-run basis.

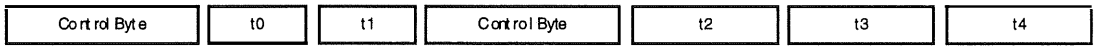


Figure 3-7 *Storage format of a ompressed texel*

Figure 3-7 shows two run lengths of different texel types. As you can see, each run length is preceded by a control byte (in this example, the different run lengths have different depths as well).

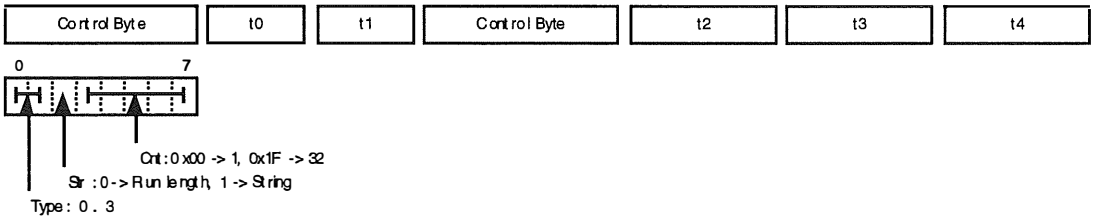


Figure 3-8 *Bits in the control byte*

Figure 3-8 shows the bits in the control byte that precedes each run length in a compressed texel. As the diagram show, a control byte has three fields. These three fields are defined as follows:

- *Type (bits 0 and 1)*—This field specifies the texel type (format) of the run of texels that immediately follow the control byte. The TYPE field is simply used to select which of the *TxtLdSrcForm* registers to choose. Four types are possible. Each type is specified by the values of the four *TxtLdSrcForm* registers. The next control byte may specify a different type for its run. All types are expanded out to the format specified within the TxtExpForm registers.
- *Str (bit 2)*—This bit specifies two kinds of storage: *string* storage when set, and *run-length* storage when cleared. When string storage is used, the CNT field (next item in this list) specifies the number of texels that follow the control

byte. When a run is copied into TRAM, each of these texels is copied to the TRAM—that is, *n* texels in the source are mapped to *n* texels in the TRAM). When run-length storage is used, the control byte is followed by only one texel. This texel is copied into the next *n* locations within the TRAM, where *n* is encoded in the CNT field (i.e. one texel in the source gets expanded to *n* texels in the TRAM).

- *Cnt (bits 3 through 7)*—This field encodes the number of texels that the current control byte will generate. If *n* is the total number of expanded texels, then *n* = CNT + 1. If STR is set to *string*, *n* also specifies the number of texels following the current control byte.

Loading a compressed texture requires two steps: Run-length decoding and texel decompression. These steps work as follows:

1. *Run-length decoding* searches for control bytes and extracts the appropriate number of texels per control byte.
2. *Texel decompression* is the process that converts the four possible source texel formats to one format that is placed in the TRAM (the TRAM may only contain one texel type).

When you store compressed texels in memory, you can use source texels that are chosen from any of the legal texel types (indexed and literal) and may be mixed to some extent within the source texture (specifically, up to four different formats can be mixed and matched per texture). The restrictions on the use is that literal and indexed texels may not be mixed within one texture.

Special Settings in the Control Byte

Two special settings can appear in the control byte shown in Figure 3-8:

- If the setting of the *Str* field is 0 and the setting of the *Cnt* field is 0, 0, the run that follows the control bit has a run length of 1. This setting is reserved, and should not be used in applications.
- If the *Type* field is programmed as transparent—that is, if *TxtLdSrcForm[TYPE].Trans = 1*— the *Str* and the *Cnt* fields are concatenated to form one 6-bit count value. No source texels follow the control byte. When a run uses a transparent format, the source texels are retrieved from a corresponding user-settable constant. See the "Texture Mapper Pipeline Register Definitions" section, later in this chapter, for more information.

If a texture in main memory is run-length encoded, it can be a mixture of the preceding formats (although literal and indexed formats cannot be mixed). The texture within the TRAM may only contain one format at a time. It is the job of the loader to ensure that only one texel type is used in the TRAM. The filtering and blending stages all use a consistent 1, 8, 8, 8, 8 format for SSB, Alpha, Red, Green and Blue.

PIP Tables

A PIP table (palette index table) is a component of the M2 Triangle Engine that is used to look up information about textures that are stored as indices. From the information stored in a PIP table, you can reconstruct the stored texels at load time.

Every PIP table has 256 entries. Each PIP-table entry is a 32-bit value constructed just like a texel. When you use a PIP table to look up information about a texture, the input texel is used as an offset into the table that contains the desired palette for the texture. The entries in a PIP table do not have to be expanded out to full 32-bit colors before filtering or texture application can be performed.

Figure 3-9 shows how a texel's color component can be used as an index into a PIP table.

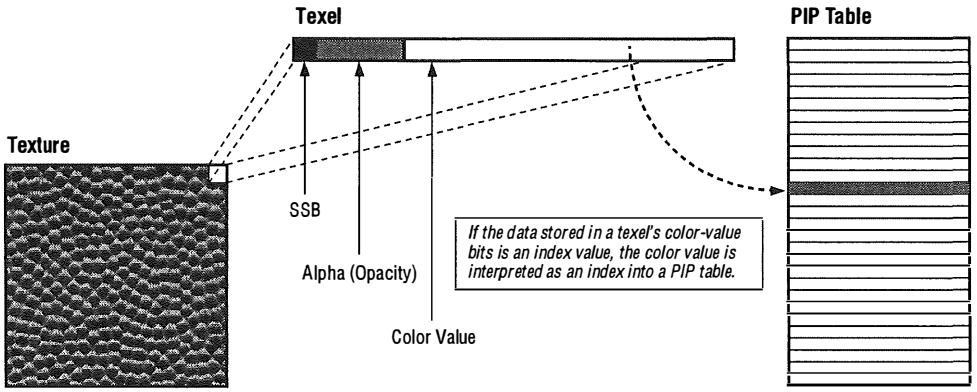


Figure 3-9 *How textures and texels work with PIP tables*

PIP tables have a fair amount of built-in flexibility. The format of input texels can range from 0 to 8 bits per texel. An alpha value may also be present. Adding an alpha value increases the number of available formats, but the alpha value is not used to reference into the PIP table. The way in which the alpha value is used depends on other bits in the texel; for details, see "How PIP Tables Work."

The Segment Pointer

Because texel formats that require less than 8 bits per texel do not make full use of the 32-bit entries provided in a PIP table, the design of the PIP table also provides a *segment pointer* that can select from a variable number of segments. For example, a PIP table can be used in 1-bpt (bit-per-texel) mode, in which there are 256 possible segments, or in 8-bpt mode, in which there is only one segment.

By using segments, you can load information for a large number of textures (or variations of textures) in a single PIP table. By using this strategy, you can avoid the need to keep unloading and reloading different PIP tables. To divide a PIP table into segments, you use a shift, a mask and a constant. You apply a shift applied to the input index, and then you use a mask to select between the shifted input index and a constant on a bit-by-bit basis.

What Can Be Stored in a PIP table

Seven bits of alpha information and one control bit (SSB) can also be stored in a PIP table. SSB, alpha, and color can each independently come from any one of these three sources:

- from the PIP Table (PIP RAM)
- from a constant
- from data directly within the source texel.

In order for any of these components to come directly from the source texel, the source texel's format must contain literal data for that component. In other words, if you want to take the alpha component from the source texel, the texture must be of a format that contains either 4 or 7 bits of alpha (see Table 3-4).

You cannot select the source texture as the output of a color value unless the input value for the color was a literal value.

How M2 Interprets Texel Data

If the color data in a texel is interpreted as a PIP offset, resulting in the texel's being passed on to a PIP table, the PIP table can also interpret the texel's color data in several ways, depending on how various texture-related attributes are set. The PIP unit can leave the texel's data values just as they are, or it can look up colors in a color table. It can also treat each of the three components in the texel as a constant. These choices provide great flexibility for certain kinds of effects.

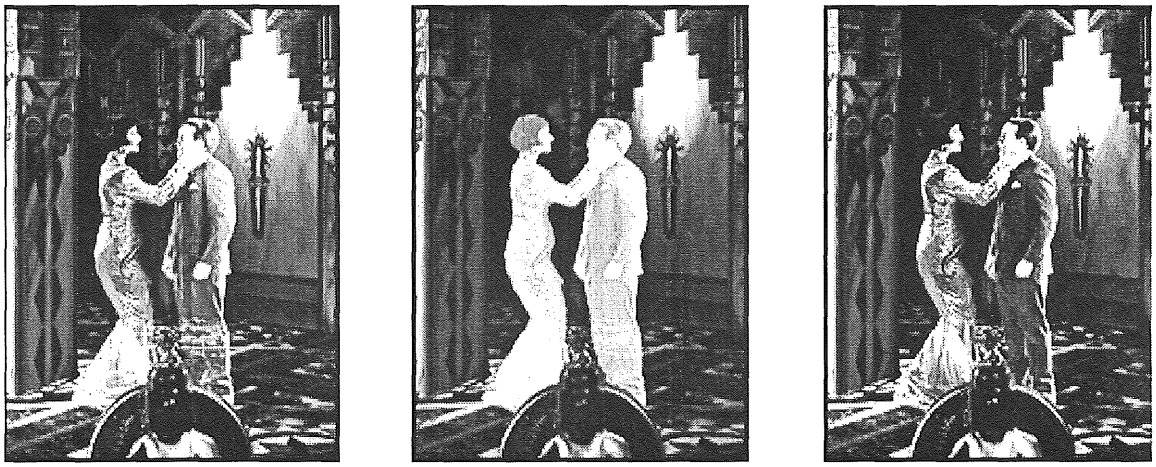


Figure 3-10 *Rendering a ghost*

How PIP Tables Work

The color, alpha, and SSB output of the PIP mapping stage can be obtained from one of three places: it can come from the texel value retrieved from TRAM, from the PIP itself, or from the two constants defined by the user. The utility of PIP mapping can be demonstrated by a simple example.

Let's look at a texture representing a ghost (see Figure 3-10). Assume that the ghost has eight colors and that you want the colors to fade in gradually over a certain number of frames. You can represent this kind of a texture can be represented using four bits per texel —three bits per texel for the color index and one bit per texel for the SSB.

In the texture-mapping example shown in Figure 3-10, the application sets color to come from the PIP table, alpha to come from the constants, and the SSB to come from the TRAM. The alpha component of the first color constant is used for texels

that have an SSB value of 0, and the alpha of the second color constant is used for texels that have an SSB value of 1. The application needs only to change the values of the color constants between different frames to fade out the ghost.

Also, the dynamic range of the alpha value can be up to seven bits. So PIP mapping can be used for certain kinds of animations while achieving significant data compression as well. The output of the PIP mapping stage is used for further processing, as described below.

The color, alpha, and SSB components of `PipConsts` are packed into a uint32 data type, as shown in Figure 3-11.

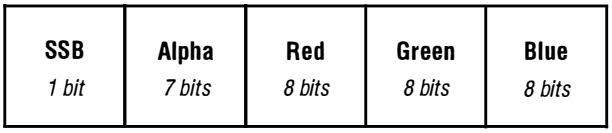


Figure 3-11 *Bit packing of a Color*

See Appendix A for an example of the register setup for a PIP load.

Texture Application Blending

The final stage of the texture pipeline is the Texture Application Blender (TAB). This is the stage when the primitive color and alpha components of each texel are blended with the filtered texel value obtained from the filtering stage. Primitive color refers to the color a pixel would have been painted in the absence of texturing. Primitive color is often the result of lighting and shading computations. The blending process for color may be composed of either a LERP or a multiply :

- LERP—$D = (1 - C) * A + C * B + f(A, B, C)$
- Multiply—$D = A * B + f(A, B)$

—where *A*, *B*, and *C* are chosen from a list of *Ct*, *At*, *Citer*, *Aiter* or constants. *f()* is an error function. *Ct* is the color of the texel (from the PIP table or from a source indexed from the PIP table; see "What Can Be Stored in a PIP table" on page 32). At is the alpha from the texel. Citer is the iterated color—that is, the interpolated color from the color data at the vertices of the triangle. *Aiter* is the iterated alpha. See Appendix A for an example of how these calculations work.

Multiplying Color and Alpha Components

When you perform LERP and multiplication operations, the input values of *A*, *B*, and *C* are assumed to have a range of [0, 1]—that is, the legal input value includes both 0 and 1. Because *A*, *B*, and *C* are represented by 8-bit numbers, the hexadecimal value 0x00 represents 0 and hex 0xFF represents 1.0.

Note that in the three application modes described under the following section "Application Modes," these values result in an error if a simple multiplication operation is performed. For example,

*0xFF * 0xFF = 0xFE*

Clearly, the result of this simple multiply should be 0xFF, not 0xFE. To compensate for this anamoly, the error function *f*() is added in. This operation applies an offset to the calculation.

Ideally, this correction should spread out the error term over the full range of output values. However, for ease of implementation, one of the inputs is passed directly to the output if the other input is '1' (0xFF). For the LERP, *A* is passed out directly if *C* is 0x00, and *B* is passed out directly if *C* is 0xFF. For the multiply, *A* is passed out directly if *B* is 0xFF, and *B* is passed out directly if *A* is 0xFF. The case of *A* and *B* both being 0xFF is left as an exercise for the reader.

The only possible function that is applied to the alpha channel is a multiply. Again, this operation passes out an unmodified value if one of the inputs is 1 (0x7F).

Application Modes

Both alpha information and color information can bypass the Texture Application Blender altogether, in which case the output may be either the textured output or the iterated output.

There are three typical application modes: *Modulate*, *Decal* and *Blend.* They are defined as follows:

- Modulate—This is used to generate effects such as a light source illuminating portions of a surface:

  $Cti = Ct * Citer; Ati = At * Aiter$

- Decal—Here the alpha in the texture map is used to superimpose a texture onto a shaded surface. The texture will not have any effect of lighting. This is useful for effects such as stenciling lettering onto the side of an object:

  $Cti = (1 - At) * Citer + At * Ct; Ati = Aiter$

- Blend—Here the color information in the texture is used to blend between a shaded surface and a background color. This is useful for effects such as transparency. For example, if we are trying to show the inside of a 3D object, such as a person, the bone would be represented by the constant color in the center and the organs surrounding the bone could be shaded, but with the bone still showing through (i.e. the organs are partially transparent):

  $Cti = (1 - Ct) * Citer + Ct * Cconst; Ati = At * Aiter$

Texture Mapping Step-by-Step

To boil it all down, M2 texture mapping requires a sequence of three operations:

1. When a particular pixel is to be painted on the screen, texel data that corresponds to the pixel data being used is retrieved from the texture. This texel data value is passed through a PIP module for further mapping. The PIP table may leave the texel data values as they are, perform a color-table lookup, or use a constant for each of the color, alpha, and SSB (source select bit) components. This strategy provides great flexibility for certain kinds of animations.
2. Texture filtering determines the level of detail (LOD) for adjacent pairs of (*U*, *V*) values to reference an appropriate texture map. It performs blending of the color/alpha values from the LODs (a) between adjacent texture maps for linear sampling, or (b) within one LOD for bi-linear sampling, or (c) a mixture of the two for quasi tri-linear sampling.

3. Texture application blending applies the texture values obtained after filtering to the color and alpha values of the triangle pixel according to the blend style specified.

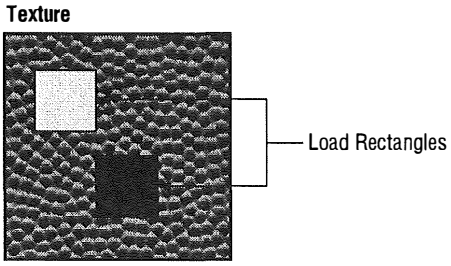


Figure 3-12 *Placing Load Rectangles inside a texture*

Loading Textures

The Texture Mapper contains a DMA engine for loading the Texture RAM (TRAM). For uncompressed textures, the maximum rate at which the TRAM can be loaded is 200MB/sec. The loader is also capable of decompressing a 3DO proprietary compressed texture format that is based around run-length encoding (RLE).

The texture loader performs four main tasks:

- *MMDMA (Memory to Memory DMA)*—This variety of load can be used to copy a certain number of bytes from main memory into the TRAM or PIP. For textures which do not require special loading operations, such as loading only a sub-rectangle of a texture in RAM into the TRAM, this load mode is the simplest to use.
- *Uncompressed Texture Load*—This kind of load is used to copy a tile from a source texture within main memory into the TRAM. The source texture must be in uncompressed format. The texel format can be any of the legal M2 formats. For uncompressed textures, a rectangular portion of a texture can be loaded.
- *Compressed Texture Load*—M2 supports run-length encoded textures. During compressed texture load, a source texture within main memory is decompressed into the TRAM.
- *PIP Load*—This type of load is used to load the contents of the PIP with a table stored in main memory.

The M2 texture loader is programmed in two stages: by first setting up the loader state and then issuing the TLD command in the TEDCntl register, or using the CLT_TxLoad( ) macro.

Both these operations can be performed in the command list. Consequently, texture loads can be performed without any overhead from the operating system.

The loader state consists of first selecting the required mode (see the preceding list) within the *TxtLdCntl* register. Each mode requires a different number of loader registers to be setup as well. MMDMA requires only three other registers to be programmed. Compressed texture load requires up to 13 registers to be programmed.

Loading/decompression does not work in parallel with the rest of the Pipeline (that is, the internal hardware pipeline that exists within the Triangle Engine). This is partly because it requires use of the same memory read and write buffers that are used by the destination blender, but it is mainly because the texture load process can swamp the main memory bandwidth while it is in operation. Registers are also shared between the loader and the main pipeline. For correct operation therefore, a SYNC instruction must be placed in the instruction list before any loader setup or the TLD instruction.

Loading Textures by Using MMDMA

MMDMA texture loads require the least amount of runtime setup of the M2 texture loader. In order to use this load mode, however, texture data has to appear in RAM exactly as it should appear in the TRAM. For example, if 3 texture maps with 4 LODs each must be loaded into the TRAM, and all of the texel data from these 12 maps fits within 16K of data, a single MMDMA load can be used to retrieve all 12 textures into the TRAM. To use MMDMA, specify the address in memory of the block of data to DMA in the TxtLdSrcAddr register, the byte count to load in the TxtCount register, and the word-aligned TRAM load offset in the TxtLdDstBase register.

Loading Uncompressed Textures

Uncompressed textures are stored as packed arrays of texels within main memory (see "How Texels Are Stored in Memory"). All the uncompressed texels stored in an array are of the same type, and the M2 hardware can automatically load a smaller tile of the source texture. Each LOD must be stored separately within main memory, and be loaded to a different base address within the TRAM.

Figure 3-13 shows how an array of uncompressed texels is stored in memory.

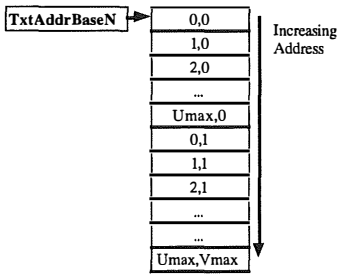


Figure 3-13 *An array of uncompressed texels ready for loading*

When an application loads an uncompressed texture, the application must point to the start of the first texel to be loaded. To support runtime texture carving (see "Runtime Carving"), and because the source texture can have any of the legal texel depths, the first texel can start on an arbitrary bit boundary. Consequently, the application must specify the bit-aligned address of the first source texel to be loaded (this might not be the same as the start of the texture itself).

The bit address of the first texel to be loaded is the concatenation of *TxtLdSrcBase* and the *SrcBitOff* field from the *TxtLdCntl* register:

{TxtLdSrcBase[7:31], SrcBitOff[0:2]}

The destination address within the TRAM must be 32-bit word-aligned, and is programmed within the *TxtLdDstBase* register. The source and destination textures can have different strides. These are programmed as bit strides in the *SrcRowBits* and *DstRowBits* fields of the *TxtLdWidth* register. In addition, the total number of rows to be loaded must be programmed in *TxtRowCnt*.

When the TLD instruction is issued, the loader first accesses the byte containing the start texel. The *SrcBitOff* offset is then applied. Next, the loader copies *TxtDstBitCnt* bits into the destination buffer and does the wrapping to re-align the data.

At the end of a line, the source memory pointer is advanced to the start of the next line. This operation is performed by advancing the source pointer by (*TxtSrcBitCnt* - *TxtDstBitCnt*). The operation is performed *TxtRowCnt* times.

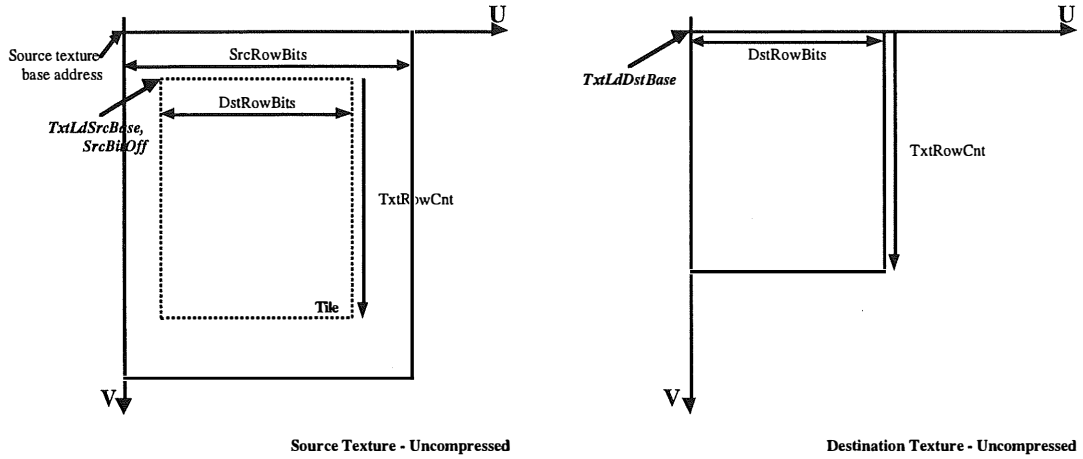


Figure 3-14 *Uncompressed Loader Setup*

See Appendix A for an example of how to load a texture.

Runtime Carving

When a texture is too large to fit within the internal TRAM, the texture (and associated geometry) must be broken into several smaller pieces. This disassembly is supported in hardware by the loader. The application sets the memory base address and also provides a *U* and *V* offset, along with the *V* dimension of the portion to load. This operation allows parts of textures to be loaded automatically.

Although runtime carving is simple in theory, there is one complication you should be aware of. The complication is that bi-linear filtering (as well as tri-linear filtering) require a 2-by-2 array of texels. For texels that are on the edge of a tile, there is no more information present to do the filtering. Hence the tiles must have a *guard region* that is one texel wide around all four edges to allow the joins in tiles to be seamless. When a guard region has been set up, you must ensure that none of the triangle vertices fall within the guard region. For more details about guard regions, see "Mipmap Filtering."

Compressed Texture Load

The M2 texture mapper decompresses textures by reading the control bytes within the compressed texture, and processes the information following each control byte. The processing consists of expanding out run-lengths and resolving different texel formats before placing the data into the TRAM.

The setup for compressed texture load is similar to that of uncompressed loads. The difference is that the texel formats are important during decompress. Also, runtime carving of compressed textures is not supported.

When loading compressed textures, the application provides the base of the compressed texture in TxtLdSrcBase, the base of the destination (uncompressed) map in TxtLdDstBase, the width of the source and destination in texels within SrcRowTex and DstRowTex, and the total number of rows to be loaded in TxtRowCnt. In addition, the texel formats must be specified as well. This is setup in the four TxtLdSrcForm registers and the TxtExpForm register. The TxtLdSrcForm registers specify the format of the four possible texel types within the source texture in main memory. The TxtExpForm register specifies the format of the texels that are to be placed into the TRAM.

See Appendix A for an example showing which registers must be programmed for a compressed texture load.

Texel Expansion

Because multiple texel formats can be present within the source texture, the formats must be expanded out to a common format before you can place them in the output buffer. The desired TRAM texel type is programmed within *TxtExpForm*. All source formats are expanded out to this type. The expansion works by first determining if each channel that is present in the output (SSB, alpha, color) is present in the source. If a certain channel is not present in the source (but is required in the output), then that data is taken from *TxtLdConst[TYPE]*, where TYPE is the two-bit index in the control byte. If the data is present in the source, but of a different depth, then the source channel is expanded as described below. The SSB, alpha and color information are handled independently, as follows:

- *SSB*—If the source texel has an SSB, that SSB is passed through to the expanded texel. If the source texel doesn't have an SSB, but the expanded texel does, a constant SSB is used (*TxtLdConst[TYPE].SSB*)
- *Alpha*—The source texel can contain 0, 4, or 7 bits of alpha information. If the expanded texel format has more than seven bits of alpha information, the source alpha must be expanded. If the source contains no alpha information, you can expand the source alpha by using a constant alpha (*TxtLdConst[TYPE].alpha*). If the source does contain alpha information, you can expand the source alpha by replicating the MSBs into the LSBs. In the case of constant alpha, either 7 bits or 4 bits are taken from the constant register, depending on the expanded format. Note that for 4-bit output, the top 4 bits of the constant alpha are chosen. Replication can be used only to go from 4 bits to 7 bits of alpha—the top three MSBs of the 4-bit source alpha are replicated into the bottom three LSBs of the expanded alpha.
- *Literal color*—A literal color can be either 555 or 888. Expansion from 555 to 888 is done by replication of MSBs. If no color is present in the source, then *TxtLdConst[TYPE].red*, *TxtLdConst[TYPE]. green*, and *TxtLdConst[TYPE].blue*

are chosen. The top five bits of the constant are chosen for 5-bit output, in much the same way that output is chosen for alpha.

- *Indexed color*—Source and expanded index depths can be [0, 1, 2, 3, 4, 5, 6,7, 8] bits. All of these can be expanded to any larger size (up to 8 bits, of course, which is the size of the PIP). The expansion is done by adding an offset (specified in *TxtLdConst[TYPE].index*) and masking to the appropriate depth. If no index is present in the source, then *TxtLdConst[TYPE].index* is used directly.

A special case occurs if the control-bit type is specified as transparent (see "Special Settings in the Control Byte"). In this case, no information follows the control byte. The appropriate color information is provided by one of the constant registers (*TxtLdConst[TYPE]*). Transparency is usually indicated by either zero SSB or alpha. The destination blender is programmed to use either of these values as a transparency flag.

See Appendix A for the register setup used when doing a transparency.

Texture Mapper Pipeline Register Definitions

This section describes the individual registers used to control the texture unit. The section provides descriptions of each of the registers, the address of the register, the CLT macro to set the register and the register layout.

The first half of this section describes the texture mapping registers, which control how a pre-loaded texture is to be interpreted and applied to geometry. The second half describes the registers used to load a texture.

Command Registers, Fields, and Macros

Each command register can be broken down into fields. For example, for a register named `XXX`, the macro `CLA_XXX` provides the data word to be written to the register with the arguments converted and packed into the register fields. The macro `CLT_XXX` takes a pointer to a pointer as the first argument. It writes the argument word to the command list and advances the pointer.

Table 3-5 *Register Memory Map*

Address	Access	Name	Information
0x000C_0000- 0x000C_3FFF	RW	TRAM	Texture RAM
0x0004_6000- 0x0004_63FF	RW	PIP	PIP RAM
0x0004_6404	RWSC	TxtLdCntl	Texture Mapper Load Control Register
0x0004_6408	RWSC	TxtAddrCntl	Address Generation Control Register / Filter modes
0x0004_640C	RWSC	TxtPIPCntl	PIP Control Register
0x0004_6410	RWSC	TxtTABCntl	Texture Application Control Register
0x0004_6414	RW	TxtLODBase0	LOD 0 Base Address
0x0004_6418	RW	TxtLODBase1	LOD 1 Base Address
0x0004_641C	RW	TxtLODBase2	LOD 2 Base Address
0x0004_6420	RW	TxtLODBase3	LOD 3 Base Address
0x0004_6424	RW	TxtLdSrcBase TxtMMSrcBase	Texture decompressor source base address MMDMA Source Base
0x0004_6428	RW	TxtCount TxtLdRowCnt TxtLdTexCnt	Byte Count for MMDMA and PIP load Row Count for uncompressed texture load Texel Count for compressed texture load
0x0004_642C	RW	TxtUVMax TxtLdWidth	Texture Size Register Texture Loader Width Register
0x0004_6430	RW	TxtUVMask	Texture Mask Register
0x0004_6434	RWSC	TxtSrcType01	Source Description Registers - Type 0 and 1
0x0004_6438	RWSC	TxtSrcType23	Source Description Registers - Type 2 and 3
0x0004_643C	RWSC	TxtExpType	Expanded Type Description Register
0x0004_6440	RW	TxtConst0 TxtPIPConst0	Source Expansion Constant Register - Type 0 PIP Constant color - SSB = 0

Address	Access	Name	Information
0x0004_6444	RW	TxtConst1 TxtPIPConst1	Source Expansion Constant Register - Type 1 PIP Constant color - SSB = 1
0x0004_6448	RW	TxtConst2 TxtTABConst0	Source Expansion Constant Register - Type 2 Texture Application Constant color - SSB = 0
0x0004_644C	RW	TxtConst3 TxtTABConst1	Source Expansion Constant Register - Type 3 Texture Application Constant color - SSB = 1

Warning: *Some registers are used for different purposes during the rendering and loading stages.*

Texture Generation Registers

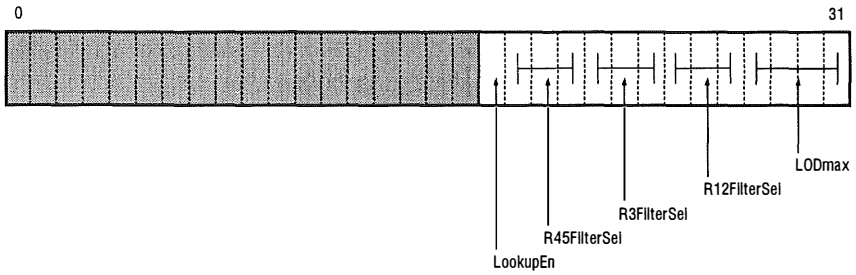
The following paragraphs list and describe the texture units' texture generation registers.

TXTADDRCNTL - Address Control Register (0x0004_6408)

The address control register specifies the way in which textures are accessed.

```
CLT_TXTADDRCNTL(pp,textureenable,minfilter,interfilter,magfilter,
    lodmax)
```

Field	Description
TEXTUREENABLE	Enables the texture lookup process
MINFILTER	Minification filter modes
POINT BILINEAR	
INTERFILTER	Inter filter modes
POINT LINEAR1 BILINEAR TRILINEAR	
MAGFILTER	Magnification filter modes
POINT BILINEAR	
LODMAX	Number of LODs present -1



The bits shown in the preceding figure are defined as follows:

- *LookupEn*—This bit enables the texture lookup process. If the *LookupEn* bit is disabled, the texture mapper does not generate any stalls to the span walker, irrespective of the texel depth and filter mode. This bit is essentially a user texture enable bit, with one exception: The texture blender is not set up to pass iterated color through by default.
- *LODmax*—Specifies the number of LODs present.

LODmax[0:3]	Definition
0x0	1 LOD
0x1	2 LODs
0x2	3 LODs
0x3	4 LODs
0x4 - 0xF	reserved

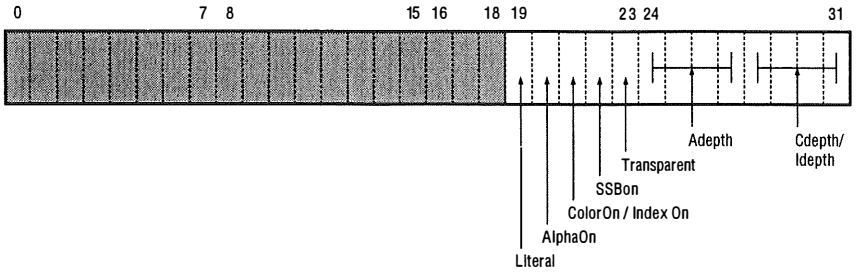
- *FilterSel*—Specifies the filter modes for the various regions. Each region may have a different filter mode selected.

FilterSel[0:2]	Definition
0x0	Point
0x1	Linear
0x2	Bilinear
0x3	Quasi-Trilinear
0x4 - 0x7	reserved

TXTEXPTYPE - Expansion Type Register (0x0004_643C)

This register describes the texel format within the texture RAM. It is used by the address generation logic to determine the depth of the texels, and by the lookup logic to unpack the data from the TRAM into SSB, color and alpha channels.

```
CLT_TXTEXPTYPE(pp, cdepth, adepth, istrans, hasssb, hascolor,
               hasalpha, isliteral)
```



The bits shown in the preceding figure are defined as follows:

- *Cdepth*—The number of bits per color component for literal formats. Only a few are valid:

Cdepth	Definition
0x0 - 0x4	reserved
0x5	5bits per color
0x6 - 0x7	reserved
0x8	8bits per color
0x9 - 0xF	reserved

- *Idepth*—The number of bits of index for indexed formats. Only the following are supported:

Idepth	Definition
0x0	reserved
0x1 - 0x8	1 - 8bits index
0x9 - 0xF	reserved

- *Adepth*—The number of bits of alpha. Only a few are valid:

Adepth	Definition
0x0 - 0x3	reserved
0x4	4bits alpha
0x5 - 0x6	reserved
0x7	7bits alpha
0x8 - 0xF	reserved

- *Transparent*—Not used during texture lookup
- *SSBon*—Specifies whether an SSB is present.

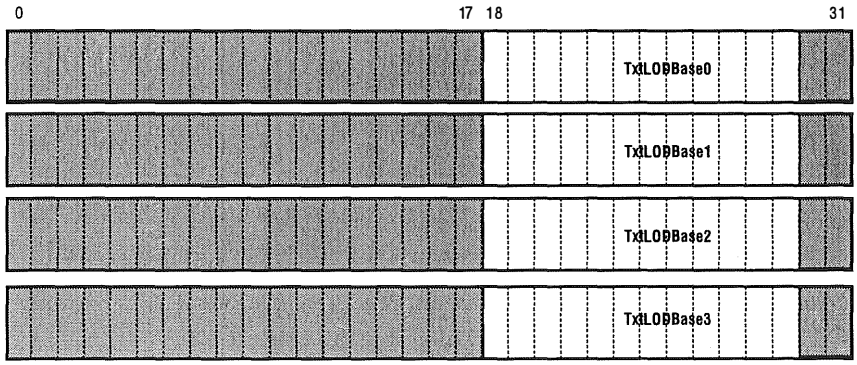
- *ColorOn, IndexOn*—Specifies whether color is present for literal texel types, or an index is present for indexed types (There are combinations that have no color bits —see below).
- *AlphaOn*—Specifies whether alpha is present.
- *Literal*—If set then the type is a literal format. If not set, then the type is indexed. See Table 3-4 for the legal texel values.

TXTLODBASE0-3 - Texture Base Registers (0x0004_6414 - 0x0004_6420)

The fields *TxtLODBase0* through *TxtLODBase0* shown in the following table are the mipmap base address registers for texture lookup. For correct operation the mipmap addresses should be 32-bit word aligned, i.e. bits 30 and 31 should be zero. This is enforced in hardware for TxtLODBase1-3, but not for TxtLODBase0, as this register is shared with the loader.

```
CLT_TXTLODBASEn (somevalue)
```

Field	Description
TXTLODBASE0	Base address for LOD0
TXTLODBASE1	Base address for LOD1
TXTLODBASE2	Base address for LOD2
TXTLODBASE3	Base address for LOD3

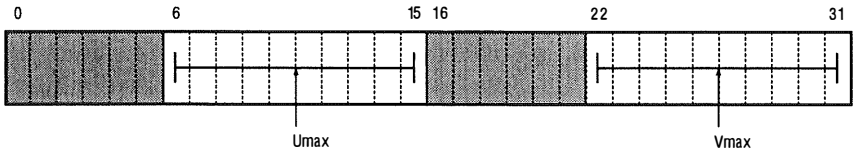


TXTUVMAX - Texture Loader Width Register (0x0004_642C)

The texture size register specifies the size of the decompressed texture during texture lookup. In this case, *UMAX* and *VMAX* (see the following table) specify the limits of the coarsest mipmap present. If LODmax is set to 0, this register refers to LOD0. If LODmax is set to 3, this register refers to LOD3. *UMAX* and *UMAX* are 10-bit quantities and should be set to the width of the texture (expressed in texels) minus 1.

```
CLT_TXTUVMAX (pp, umax, vmax)
```

Field	Description
UMAX	Width of destination buffer in bits
VMAX	Width of source buffer in bits

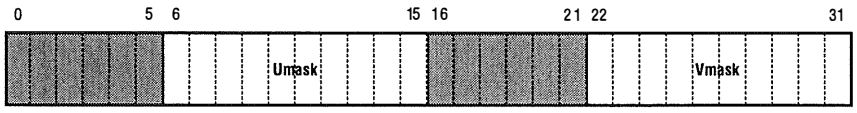


TXTUVMASK - Texture Mask Register (0x0004_6430)

The texture mask is used for texture replication. Bits set to '0' in this register will be masked off during address calculation. For normal operation, this register should be programmed with 0x3FF for both masks. Note that for tiling to work properly as a result, the source texture must be a power of two in U and V. See Appendix A for the register setup used to tile a texture.

```
CLT_TXTUVMASK (pp, umask, vmask)
```

Field	Description
UMASK	Enables mask
VMASK	Enables mask

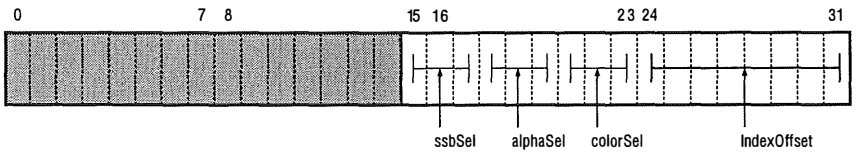


TXPIPCNTL - PIP Control Register (0x0004_640C)

This register controls the operation of the PIP circuitry. For the constant, the input SSB from the TRAM lookup chooses between *TxtPIPConst0* and *TxtPIPConst1* registers. Note that if *colorSel* is 0x1 (meaning that the PIP RAM is bypassed), the SSB or Alpha output should be chosen from either options 0x0 or 0x1 above for correct operation. If there is no SSB in the source texture, the Lookup section passes SSB as 0—so, in this case, *TxtPIPConst0* is always selected .

```
CLT_TXPIPCNTL (pp, pipssbselect, pipalphaselect, pipcolorselect,
   pipindexoffset)
```

Field	Description
PIPSSBSELECT	Select PIP SSB module output
CONSTANT TEXTURE PIP	
PIPALPHASELECT	Select PIP Alpha module output
CONSTANT TEXTURE PIP	
PIPCOLORSELECT	Select PIP color module output
CONSTANT TEXTURE PIP	
PIPINDEXOFFSET	Index offset



The bits shown in the preceding figure are defined as follows:

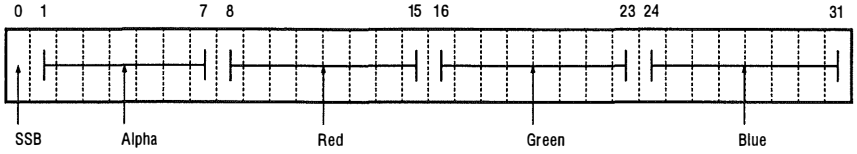
- *IndexOffset*—This field is added to the input index before accessing into the PIP RAM. This can be seen as a base pointer from where PIP accesses are to occur. The generated index wraps from 255 -> 0 on an overflow.
- *ssbSel, alphaSel, colorSel*—These fields specify the output of the PIP module for SSB, alpha and color.

ssbSel, alphaSel	Definition
0x0	Constant
0x1	Texture Cache
0x2	PIP
0x3 - 0x7	reserved

PIP Constant Registers (0x0004_6440 - 0x0004_6444)

The SSB from the TRAM Lookup is used to select between these two registers. The output for SSB, alpha or color can be independently controlled to be from the selected register. Note that if no SSB is present in the source texture, then TxtPIPConst0 is always selected.

```
CLT_TXTCONSTn (pp, blue, green, red, alpha, ssb) n = 0,1
```



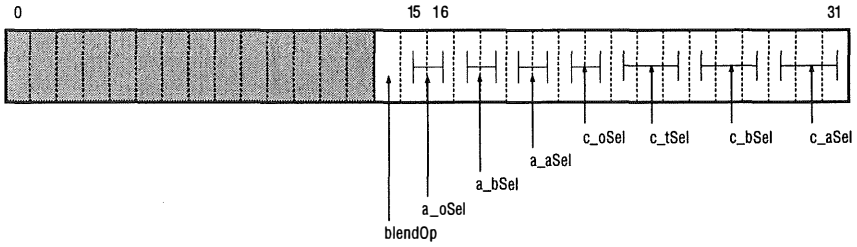
TXTTABCNTL - Texture Application Control Register (0x0004_6410)

This register controls the data path of the texture application blender. It controls the input selects to the color LERP function, and the output selects for the alpha and the color channels independently.

```
CLT_TXTTABCNTL (pp, firstcolor, secondcolor, thirdcolor, firstalpha,
    secondalpha, colorout, alphaout, blendop)
```

Field	Description
FIRSTCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select first color input
SECONDCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select second color input
THIRDCOLOR PRIMALPHA PRIMCOLOR TEXALPHA TEXCOLOR CONSTALPHA CONSTCOLOR	Select third color input
FIRSTALPHA PRIMALPHA TEXALPHA CONSTALPHA	Select first alpha input
SECONDALPHA PRIMALPHA TEXALPHA CONSTALPHA	Select second alpha input

Field	Description
COLOROUT	Select color output
PRIMCOLOR TEXCOLOR BLEND	
ALPHAOUT	Select Alpha output
PRIMALPHA TEXALPHA BLEND	
BLENDOP	Control the LERP function



The bits shown in the preceding figure are defined as follows:

- *c_aSel, c_bSel, c_tSel*—Control the inputs to the color LERP function.

c_aSel, c_bSel, c_tSel	Definition
0x0	Aiter
0x1	Citer
0x2	At
0x3	Ct
0x4	Aconst
0x5	Cconst
0x6 - 0x7	reserved

Note: *The constant refers to the two constant registers: TxtCATIConst0 and TxtCATIConst01. They are selected by the SSB coming into the blender from the filter.*

- *a_aSel, a_bSel*—Control the inputs to the alpha Multiplier function.

a_aSel, a_bSel	Definition
0x0	Aiter

a_aSel, a_bSel	Definition
0x1	At
0x2	Aconst
0x3	reserved

- *c_oSel, a_oSel*—These fields control the datapath mux at the output from the blender. The two are independent.

c_oSel	Definition
0x0	Citer
0x1	Ct
0x2	Blend output
0x3	reserved

a_oSel	Definition
0x0	Aiter
0x1	At
0x2	Blend output
0x3	reserved

Note: *If options 0 or 1 are selected above, then the inputs to the LERP block and alpha multiply are don't care.*

- *blendOp*—Controls the function of the LERP.

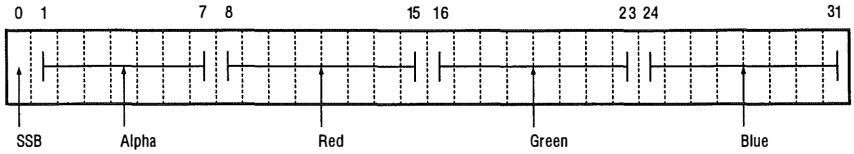
blendOp	Definition
0x0	LERP (A*(1-t) + B * t)
0x1	MULT (A * B)

Note: *For the MULT function, the t input to the LERP block is don't care.*

TAB Constant Registers (0x0004_6448 & 0x0004_644C)

The SSB from the output of the filter is used to select between these two registers. The output for SSB, alpha or color from the TAB can be independently controlled to be from the selected constant register or from the pipeline.

```
CLT_TXTCONSTn (pp, blue, green, red, alpha, ssb) n = 2, 3
```



Texture Loader Register Definitions

The texture loader registers are defined in the following paragraphs.

TXTCONST (0, 1, 2, 3) - Source Expansion Constant Registers (0x0004_6440 - 0x0004_644C)

These four sets of constants registers are used during the decompression of compressed textures to generate constant colors or index for each of the four source texture texel types. `TXTCONST0` and `TXTCONST1` select the PIP constant. `TXTCONST2` and `TXTCONST3` select the texture application constant. For indexed types, the *Blue* field can be used as an offset that is added to the input index value. If no index value is provided, then several bits (depending on the color depth) may be taken as an index constant.

Note that when 5 bits of color is selected for expansion, the 5-bit color is right-aligned into the 8-bit channel. Likewise, 4-bit alpha is right-aligned into the 7-bit alpha channel.

The SSB from the TRAM Lookup is used to select between these two registers. The output for SSB, alpha or color can be independently controlled to be from the selected register.

`CLT_TXTCONSTn (pp, blue, green, red, alpha, ssb)`

Field	Description
BLUE	Select Blue value
GREEN	Select Green value
RED	Select Red value
ALPHA	Select Alpha value
SSB	Select SSB value

TXTSRCTYPE*n* - Texture Source Types (*0x0004_6434 - 0x0004_643c)*

These register are used during texture load of compressed textures only. There are five format description registers—one for each of the four compressed texel types within the source texture (0, 1, 2, 3), and one for the expanded format that is used to be loaded into the TRAM.

```
CLT_TXTSRCTYPEn (pp, cdepthn, adepthn, istransn, hassssbn,
                 hascolorn, hasalphan, isliteraln)
```

Field	Description
CDEPTH(0, 1, 2, 3)	Number of bits per color component for literal formats (0x5=5 bits per color, 0x8=8 bits per color)
ADEPTH(0, 1, 2, 3)	Number of Alpha bits (0x4=4 bits Alpha, 0x7=7 bits)
ISTRANS(0, 1, 2, 3)	Selects constant Alpha or SSB for run of compressed texels
HASSSB(0, 1, 2, 3)	Specifies if SSB is present
HASCOLOR(0, 1, 2, 3)	Specifies if color is present for literal texel types
HASALPHA(0, 1, 2, 3)	Specifies if Alpha is present
ISLITERAL(0, 1, 2, 3)	If set, the type is a literal format

```
CLT_TXTEXPTYPE(pp, cdepth, adepth, istrans, hasssb, hascolor,
               hasalpha, isliteral)
```

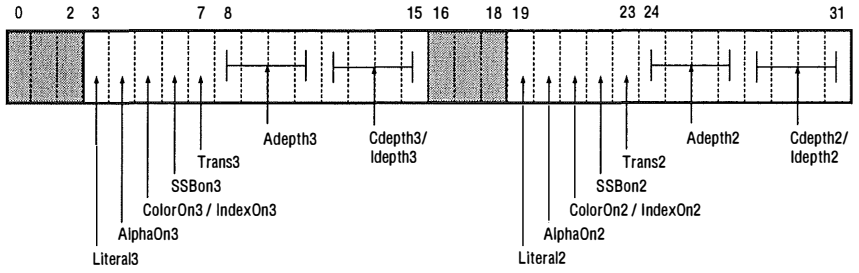


Figure 3-15 *TxtLdSrcType01*

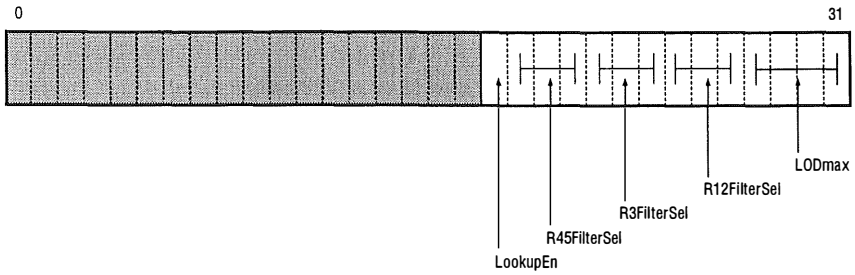


Figure 3-16 *TxtLdSrcType23*

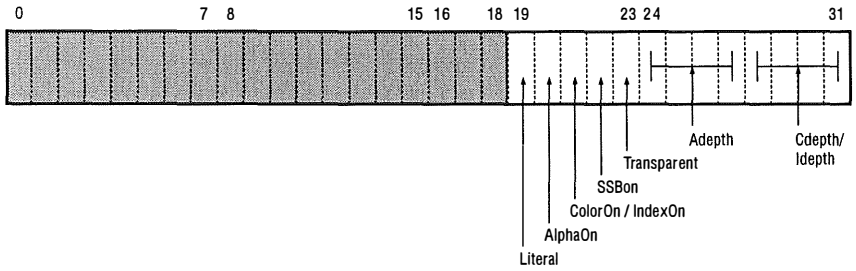


Figure 3-17 *TxtExpType*

These registers are used during texture load of compressed textures only. There are five format description registers—one for each of the four compressed texel types within the source texture, and one for the expanded format that is used to be loaded into the TRAM.

The fields specify the following information:

- *Cdepth*—Literal formats are not supported in compressed textures.
- *Idepth*—The number of bits of index for indexed formats. Only the following are supported:

Idepth	Definition
0x0	reserved
0x1 - 0x8	1 - 8bits index
0x9 - 0xF	reserved

- *Adepth*—The number of bits of alpha. Only a few are valid:

Adepth	Definition
0x0 - 0x3	reserved
0x4	4 bits alpha
0x5 - 0x6	reserved

Adepth	Definition
0x7	7 bits alpha
0x8 - 0xF	reserved

- *Transparent*—Used only for compressed texture source description. If this field is set, the decompressor uses a constant color, alpha or SSB for that run length of compressed texels. The constant register is selected from one of the four *TxtSrcConstN* registers, where *N* indicates the type of the current control byte within the decompressor.
- *SSBon*—Specifies whether an SSB is present.
- *ColorOn*, *IndexOn*—Specifies whether index color is present. Note that there are combinations that have no color bits.
- *AlphaOn*—Specifies whether alpha is present.
- *Literal*—If this field is set, the type is a literal format. If this field is not set, the type is indexed.

  The decompressor can support all the indexed pipeline formats. Literal pipeline formats can not be compressed. The texels are decompressed at different rates depending on their total depth. The following table shows the complete set of supported texels and the speed at which they can be loaded.

Color	Alpha	SSB	Total	Speed
0	-	1	1	4
1	-	-	1	4
1	-	1	2	4
2	-	-	2	4
0	4	-	4	4
3	-	1	4	4
4	-	-	4	4
6	-	-	6	2
5	-	1	6	2
1	4	1	6	2
2	4	-	6	2
0	7	1	8	2
3	4	1	8	2
4	4	-	8	2
7	-	1	8	2
8	-	-	8	2

Color	Alpha	SSB	Total	Speed
4	7	1	12	1
7	4	1	12	1
8	4	-	12	1
8	7	1	16	1

In the preceding table, the speed column specifies the number of texels loaded per tick. The speed is independent of the source type—it depends only on the expanded type to be written to the TRAM.

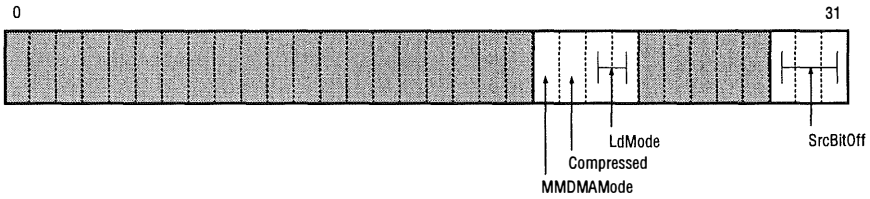
Note also that compressed source texels cannot contain more data for each of the data types (color, alpha, SSB) than the expanded format. They can only contain an equal amount or less (none).

TXTLDCNTL - Texture Loader Control Register (0x0004_6404)

This register controls the texture load process. When the TLD instruction is issued, this register is used to determine which type of load function to select.

```
CLT_TXTLDCNTL (pp, compressed, loadmode, srcbitoff)
```

Field	Description
COMPRESSED	Specifies that the source texture in memory is compressed and needs decompression before use.
LOADMODE LOADMODE_TEXTURE LOADMODE_MMDMA LOADMODE_PIP	Determines the operation of the memory-to-memory DMA process.
SRCBITOFF	Source bit offset used with uncompressed texture loading.



The bits shown in the preceding figure are defined as follows:

- *LdMode*—Specifies the mode of the texture loader.

LdMode[0:1]	Definition
0x0	Texture Load
0x1	MM DMA
0x2	PIP Load
0x3	reserved

Note: *When MMDMA is selected, the target is determined by the MMDMATramOn and MMDMAPipOn bits within the supervisor TxtCntl register.*

- *Compressed*—If this bit is enabled, it indicates that the source texture in memory is compressed and must be decompressed before it is used. This bit is used only if *LdMode = 0* (*i.e.*, Texture Load).
- *SrcBitOff*—This field is used by the texture loader during the loading of uncompressed textures (*LdMode = 0 && Compressed== 0*). It indicates the bit start position within a byte. This is used in conjunction with *TxtLdSrcBase* to define the exact bit start position of a texel in memory.

TXTLDSRCOFFSET - Text Load Source Offset Register

This register provides the compressed texture start offset value.

```
CLT_TXTLDSRCOFFSET (somevalue)
```

Field	Description
VALUE	Select offset value

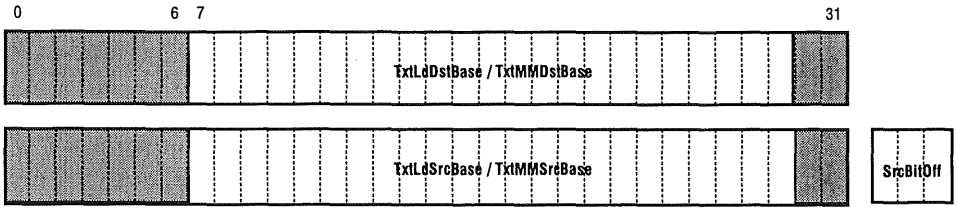
TXTLDSRCADDR- Loader Base Registers(0x0004_6424 & 0x0004_6414)

TxtLdSrcBase describes the byte-aligned source address in main memory. Compressed textures may start on any byte aligned address. Uncompressed textures may start on any bit address. The *SrcBitOff* field within the *TxtLdCntl* register is effectively used as three extra MSBs for uncompressed load. PIP contents are byte aligned as well as MMDMA source addresses.

```
CLT_TXTLDSRCADDR(pp, x)
```

TxtLdDstBase describes the 32-bit word aligned destination address within the TRAM. The base address of all load functions will be on a 32-bit boundary. This is because the TRAM logic does not support a RMW cycle into the RAM to allow for arbitrary alignment

```
CLT_TXTLODBASE0(pp, x)
```



TXTCOUNT - Texture Loader Count Register (*0x0004_6428*)

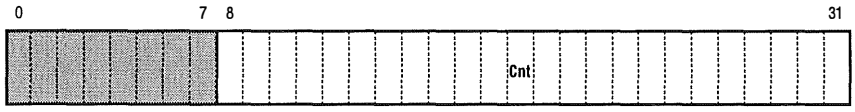
This register specifies different counts for the various loader modes:

- During uncompressed texture load, TxtLdRowCnt indicates the number of rows of texels to copy into the TRAM.
- During compressed texture load, TxtLdTexCnt indicates the total number of texels to decompress.
- During MMDMA, TxtLdByteCnt indicates the number of bytes to copy.
- During PIP load, TxtLdByteCnt indicates the number of bytes to copy into the PIP.

```
CLT TXTCOUNT (somevalue)
```

Field	Description
TXTCOUNT	During compressed texture load, indicates the total number of texels to decompress. During compressed texture load, indicates the total number of texels to decompress. During MMDMA, indicates the number of bytes to copy. During PIP load, indicates the number of bytes to copy into the PIP.

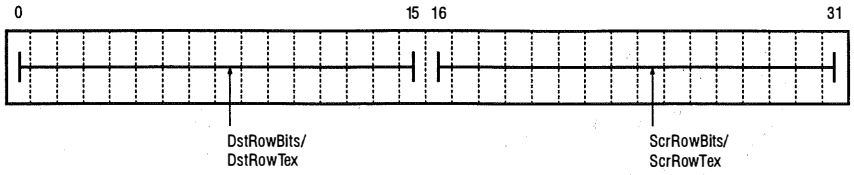
Note that a count of 0 does mean zero—that is, "Do something zero times."



TxtUVMax-Texture Loader Width Register (0x0004_642C)

During texture load of uncompressed textures, this register indicates the width of the destination and source buffers in bits. A count of 0 indicates a width of zero.

```
CLT_TXTUVMAX(pp, umax, vmax)
```

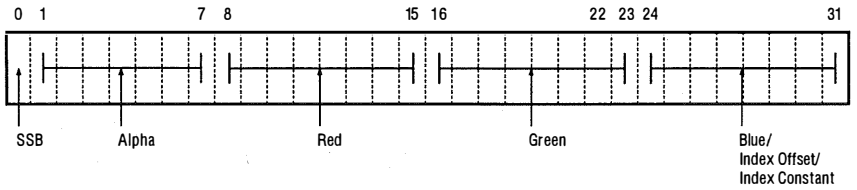


TxtConst2-3-Decompress Constant Registers (0x0004_6448 - 0x0004_644C)

M2 provides two constant registers. They are used only during decompresses of compressed textures. They are used to generate constant colors or index for each of the four source texture texel types. For indexed types the *Blue* field can be used as an offset which is added to the input index value. If no index value is provided, then several bits (depending on the color depth) may be taken as an index constant.

```
CLT_TXTCONSTn(pp, red, green, blue, alpha, ssb) n = 2,3
```

Note that when 5 bits of color are selected for expansion, the 5-bit color is right-aligned into the 8-bit channel. Likewise, 4-bit alpha is right-aligned into the 7-bit alpha channel.



Chapter 4

Destination Blender

Once pixels have been passed through the Texture Mapper, they pass into the Destination Blender, where the last four processing steps take place:

- Pixels can be optionally discarded from a triangle, based on several criteria.
- Pixels can be blended with a color, or pixels from another frame buffer.
- Pixels can be dithered.
- Pixels can be Z-buffered, so that they are only drawn if they are from visible portions of a triangle that intersects other triangles.

Finally, the pixels are written to the frame buffer.

This chapter contains the following topics:

Pixel Discards

In the Destination Blender, pixels can be discarded before being output to the frame buffer or Z-buffer. This can be useful, for example, to mask away unwanted parts of a triangle that fall outside of the shape of a sprite that is texture-mapped onto a pair of triangles. Discards can be set to happen whenever any combination of the following events occur:

- Alpha of the pixel is 0.0
- Color of the pixel is pure black (red=0.0, green=0.0, blue=0.0)
- SSB of the pixel is 0
- Pixel falls outside the area covered by the Z-buffer

Note: *Since the pixels enter the Destination Blender after being passed through the Texture Mapper, the color, alpha, and SSB used for discards are the ones that come from the Texture Blender.*

The Triangle Engine only allows you to allocate a Z-buffer that is smaller than the frame buffer. The Z-discard allows pixels that fall outside of the Z-buffered area to be immediately discarded. For more information, see the "Z-buffering" section, later in this chapter.

There is no performance hit when discards are enabled, and they can allow better performance in some cases, since discarded pixels are not passed into the Z-buffering unit or pixel blender.

Pixel Blending

After the discard process is complete, the remaining pixels are passed to a sophisticated blender, where the color can be blended with a color from any of a number of sources, including a second source frame buffer. This blender is typically used to make pixels partially transparent, or to globally tint a scene toward a constant color, such as in a screen fade. Because the blend can be weighted based on alpha, carefully constructed triangles can use the Destination Blender to apply fog and some other special effects to the scene.

The two colors to be blended (we're calling them A and B) can be blended according to the following equation:

$$Cout = ((MA \bullet A)\ op\ (MB \bullet B))\ <<\ shift$$

A, B, MA, and MB can come from a variety of sources. The following table describes the different combinations that can be used for these variables:

Variable	Possible Sources
A	Color of the pixel from the Texture Mapper Constant color Alpha of the pixel from the Texture Mapper Complement of the color from the source frame buffer

Variable	Possible Sources
B	Color from a source frame buffer Constant color Alpha of the pixel from the Texture Mapper Complement of the color of the pixel from the Texture Mapper
MA	Alpha of the pixel from the Texture Mapper Alpha from the source frame buffer One of a pair of constants (selected by the SSB) Color of the pixel from the source frame buffer
MB	Alpha of the pixel from the Texture Mapper Alpha from the source frame buffer One of a pair of constants (selected by the SSB) Color of the pixel from the Texture Mapper

Besides the flexibility that the various selectors for A, B, MA, and MB provide, you can also decide what operation you want to perform. The operation can be an add or subtract, with or without clamping, or any boolean operator, such as AND, OR, etc.

Finally, color components can be bit-shifted left or right up to 3 bits. The shift is actually always a left shift, but the parameter allows for a shift left by -3 bits, which results in a right shift 3 bits.

When the pixel blender is enabled, pixels can be output at a maximum throughput of 1 pixel per tick.

Pixel Scaling

The Triangle Engine performs color calculations on 8-bit numbers for each channel. Source images, as well as destination frame buffers, can be 16- or 32-bit numbers. The Triangle Engine converts the 16-bit pixel values to 32-bit values before any computations are performed.

You have an option to right-shift by three bits the input source and/or texture 8-bit values. In this way, your application can combine as many as eight images without overflowing the pixel values. You can use this capability to accumulate images in the destination frame buffer. Resulting accumulated images are then scaled appropriately. Attributes associated with this capability are the `DblARightJustify` attribute, the `DblBRightJustify` attribute, and the `DblFinalDivide` attribute.

Source Blending

By blending pixels with a source frame buffer, you can easily produce effects like transparency and reflections. But, you need to set up the correct registers in order to ensure correct output. The M2 Triangle Engine needs to know the source frame buffer's address, width, height, pixel depth, and stride. The source frame buffer can be the same bitmap that is used as the destination frame buffer.

When blending with a source frame buffer is enabled, pixels can be output at a maximum rate of 1 pixel every other tick.

Dithering

The M2 Triangle Engine can dither pixels before they are sent to the destination frame buffer. This reduces the visual artifact of mock banding when 16-bit frame buffers are used. Here's how the ditherer works:

You load a 4 x 4 matrix of dither values. Valid matrix values are 4-bit signed integers, between -8 and +7. The matrix is packed into two 32-bit registers. One register contains the matrix values for the top two rows, the other the bottom two rows.

The least significant two bits of the x- and y-coordinates of the pixel are used as indices into the 4 x 4 matrix. The value from the matrix is retrieved, and added to the red, green, and blue components of the pixel.

Dithering can be performed at a rate of 2 pixels per tick.

See Appendix A for the register setup used with dithering.

Z-buffering

The M2 Triangle Engine supports Z-buffering for pixels that contain depth information (triangles which have a W value). By using a Z-buffer, you can render objects in any order, and have them intersect each other, without worrying about how overlapping regions are drawn. Normally, the Z-buffer only draws pixels that are closer to the viewer than previously-rendered pixels, but it can be set to work in other modes.

The Z-buffer works by storing depth information in a special 16-bit frame buffer. The contents of each 16-bit pixel are in a proprietary 3DO format. At the beginning of a frame, the best way to clear the Z-buffer is to render two large triangles with a W value of 0.0, and set the Z-bufferer to force updates to the Z-buffer, regardless of what the value is. For more information about modifying the operating mode of the Z-buffer, see the description of the DBZCNTL register in the "Destination Blender Register Definitions" section, later in this chapter.

It is not necessary to allocate a Z-buffer to be the size of the entire frame buffer. In some cases, such as an application where the top portion of the screen is used for player's status, it might only be necessary to allocate a Z-buffer to be the size of the playfield area of the screen, thus saving some memory.

Note: *Although the Z-buffer does not need to be as tall as the destination frame buffer, it must be as wide as the destination frame buffer's stride.*

Z-banding

One other technique that can save some RAM (but at the cost of some performance) is to allocate a Z-buffer that is a fraction of the height of the frame buffer. This technique is referred to as *Z-banding*. Here's how Z-banding works: The Z-buffer is aligned with the top of the destination frame buffer. The scene is rendered and the Z-buffer is moved down. The scene is re-rendered and the Z-buffer moved down again. This process is repeated until the whole frame buffer is drawn. Using this technique, in combination with the Destination Blender's discard when pixels are outside the region covered by the Z-buffer, allows Z-buffering to occur when there is not enough RAM for an entire Z-buffer.

However, because the Triangle Engine needs to pass over the data multiple times, performance will usually not be as good as the case where a complete Z-buffer can be allocated.

When Z-buffering is enabled, performance is 2 pixels per tick if the pixels do not need to update the frame buffer or Z-buffer, and 1 pixel per tick in the case where the pixel needs to be drawn.

See Appendix A for the register setup used with Z-buffering.

Window Clipping

An application can specify a rectangular window for clipping out the pixels that lie on the inside or the outside of this rectangle. The rectangle should be specified in conjunction with the `DblEnableAttrs` attributes of `WinClipInEnable` or `WinClipOutEnable`.The *wWindow* rectangle is specified with `DblXWinClipMin`, `DblXWinClipMax`, `DblYWinClipMin`, and `DblYWinClipMax`.

Destination Blender Register Definitions

This section describes the individual registers used to control the Destination Blender. Each command register can be broken down into fields. For a register named XXX, the macro `CLA_XXX` provides the data word to be written to the register with the arguments converted and packed into the register fields. The macro `CLT_XXX` takes a pointer to a pointer as the first argument. It writes the argument word to the command list and advances the pointer.

This section provides descriptions of each of the registers, the address of the register, the CLT macro to set the register and the register layout.

Warning: *Some registers are used for different purposes during the rendering and loading stages.*

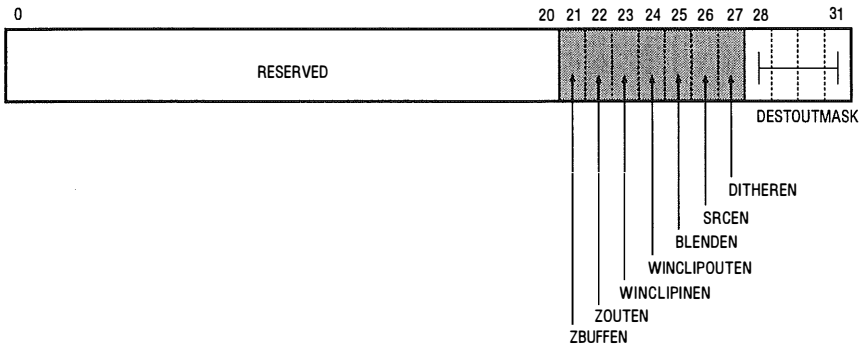
DBUSERCONTROL - User Destination Blend Control Register (0x0004_8008)

This register provides general control of the destination blend logic.

```
CLT_DBUSERCONTROL(pp, zbuffen, zouten, winclipinen, winclipouten,
                  blenden, srcen, ditheren, destoutmask)
```

Field	Description
ZBUFFEN	Enable Z-buffering.
ZOUTEN	Enable Z-buffer output.
WINCLIPINEN	Enable window clipping inside the clip range.
WINCLIPOUTEN	Enable window clipping outside the clip range.
BLENDEN	Enable color, Alpha, and DSB blending.
SRCEN	Master control for the source input. Enables color, Alpha, and DSB blending.

Field	Description
DITHEREN	Enable dithering for output to 555.
DESTOUTMASK	Enable byte output (DSB, RGB, Alpha)

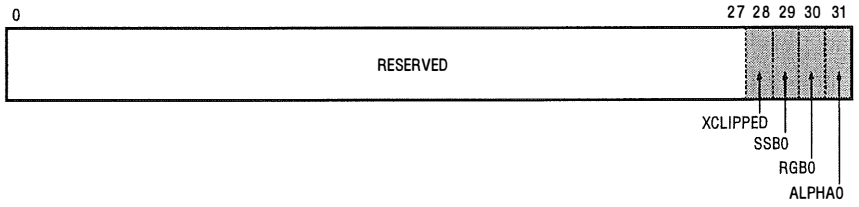


DBDISCARDCONTROL - Pixel Discard Control Register (0x0004_800c)

This register provides pixel discard control.

```
CLT_DBDISCARDCONTROL (pp, zclipped ssb0, rgb0, alpha0)
```

Field	Description
ZCLIPPED	Enable pixel discard when outside of the buffer region.
SSB0	Enable pixel discards based on SSB.
RGB0	Enable pixel discards if R == G == B == 0.
ALPHA0	Enable pixel discards if Alpha == 0.



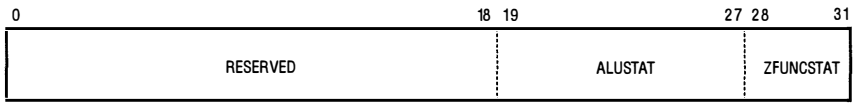
DBINTCNTL - Interrupt Control Register (0x0004_8014)

This register provides masks to control which ALU and Z results can generate

interrupts to the CPU.

```
CLT_DBINTCNTL (PP, ALUSTAT, ZFUNCSTAT)
```

Field	Description
ALUSTAT	AND'd with ALU status bits and OR'd to create ALUStatInt
ZFUNCSTAT	AND'd with Z function status bits and OR'd to create ZFuncInt

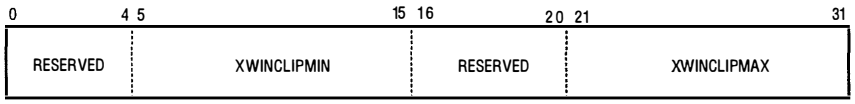


DBXWINCLIP - X Window Clip Value Register (0x0004_801c)

This register defines the X border of the user-defined render window.

```
CLT_DBXWINCLIP (pp, xwinclipmin, xwinclipmax)
```

Field	Description
XWINCLIPMIN	X window clip minimum (left)
XWINCLIPMAX	X window clip maximum (right)

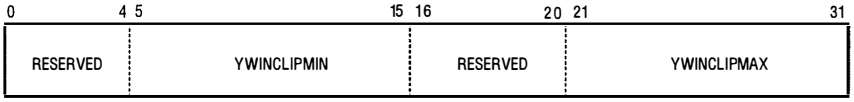


DBYWINCLIP - Y Window Clip Value Register (0x0004_8020)

This register defines the Y border of the user-defined render window.

```
CLT_DBXWINCLIP (pp, xwinclipmin, xwinclipmax)
```

Field	Description
YWINCLIPMIN	Y window clip minimum (left)
YWINCLIPMAX	Y window clip maximum (right)

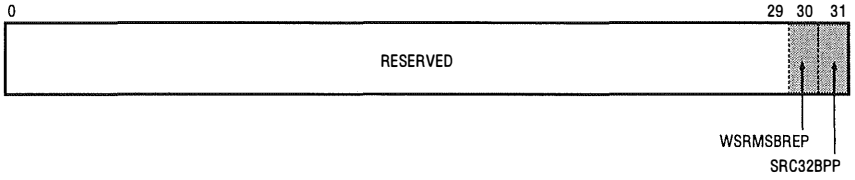


DBSRCCNTL - Source Read Control Register (0x0004_8030)

Specifies the format of the buffer to be used for source blending.

```
CLT_DBSRCCNTL (pp, srcmsbred, src32bpp)
```

Field	Description
WSRMSBREP	For 16 Bpp, Replicate 3 MSBs into 3 LSB's if left justified (otherwise insert zeros).
SRC32BPP	Source pixel format (16 or 32 bpp).

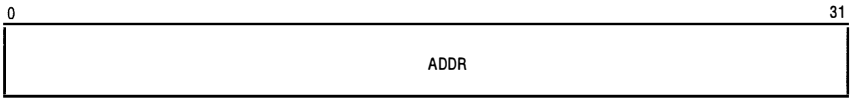


DBSRCBASEADDR - Source Base Address Register (0x0004_8034)

This register provides the source bitmap address when the destination blender's source blending mode is enabled.

```
CLT_DBSRCADDR (pp, addr)
```

Field	Description
ADDR	Base address of source buffer



DBSRCXSTRIDE - Source X Stride Register (0x0004_8038)

This register contains the source read X stride value - the distance in pixels to the

next line.

```
CLT_DBSRCXSTRIDE (pp, xstride)
```

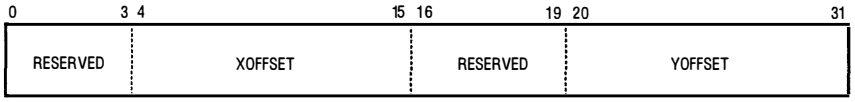
Field	Description
XSTRIDE	X Stride value.

DBSRCOFFSET - Source Offsets Register (0x0004_803c)

This register enables the X and Y offsets used for source reads.

```
CLT_DBSRCOFFSET (pp, xoffset, yoffset)
```

Field	Description
XOFFSET	X offset
YOFFSET	Y offset

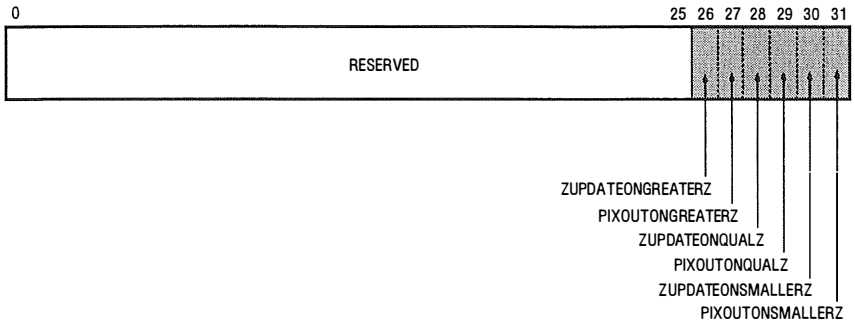


DBZCNTRL - Z Buffer Control Register (0x0004_8040)

This register defines the ZPixOut and ZBufOut results.

```
CLT_DBZCONTROL (pp, zupdateongreaterz, pixoutongreaterz,
                zupdateonqualz, pixoutonqualz, zupdateonsmallerz,
                pixoutonsmallerz)
```

Field	Description
ZUPDATEONGREATERZ	If new Z greater than current Z, update Z
PIXOUTONGREATERZ	If new Z greater than current Z, update pix
ZUPDATEONQUALZ	If new Z equal to current Z, update Z
PIXOUTONQUALZ	If new Z equal to current Z, update pix
ZUPDATEONSMALLERZ	If new Z smaller than current Z, update Z
PIXOUTONSMALLERZ	If new Z smaller than current Z, update pix

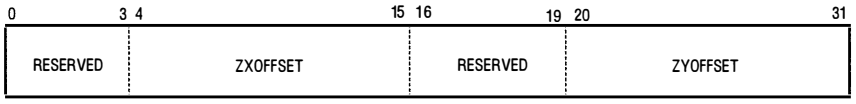


DBZOFFSET - Z Offset Register (0x0004_8048)

This register contains the X and Y offsets used for Z.

```
CLT_DBZOFFSET (pp, zxoffset, zyoffset)
```

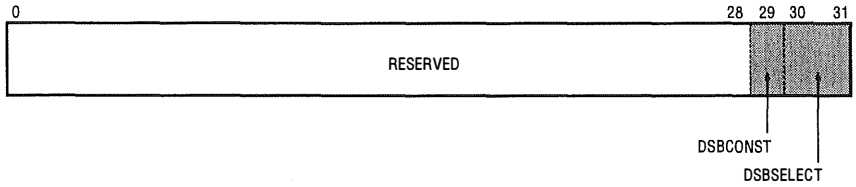
Field	Description
ZXOFFSET	X offset for Z
ZYOFFSET	Y offset for Z



DBSSBDSBCNTL - SSB/DSB Control Register (0x0004_8050)

```
CLT_DBSSBDSBCNTL (pp, dsconst, dbselect_xxxxxxx)
```

Field	Description
DSBCONST	DSB constant
DSBSELECT_	Select DSB generation:
OBJSSB	Use SSB (0)
CONST	Use constant (1)
SRCSSB	Use source input (2)

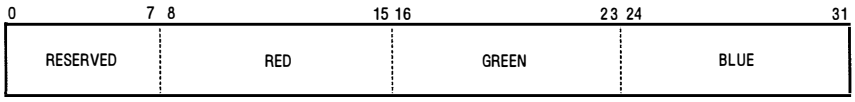


DBCONSTIN - Constant In Register (0x0004_8054)

This register contains the RBG constant used for computation input.

```
CLT_DNCONSTIN (pp, red, green, blue)
```

Field	Description
RED	Red constant
GREEN	Green constant
BLUE	Blue constant



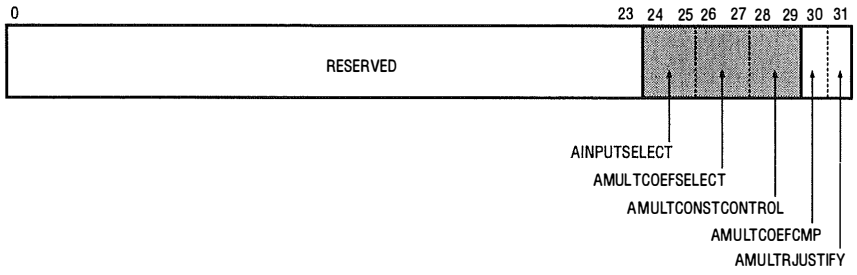
DBAMULTCNTL - Texture Multiplication Control Register (0x0004_8058)

This register controls the texture color multiplication.

```
CLT_DBAMULTCNTL (ainputselect, amultcoefselect,
                 amultconstcontrol, amultrjustify)
```

Field	Description
AINPUTSELECT	Choose texture, constant, or 1-Cs for texture input: TEXCOLOR (0) CONSTCOLOR (1) SRCCOLORCOMPLEMENT (2) TEXALPHA (3)

Field	Description
AMULTCOEFSELECT	Choose texture coefficient: TEXALPHA (00) SRCALPHA (08) CONST (10) SRCCOLOR (18) TEXALPHACOMPLEMENT (01) SRCALPHACOMPLEMENT (09) CONSTCOMPLEMENT (11) SRCCOLORCOMPLEMENT (19)
AMULTCONSTCONTROL	Choose constant controller: TEXSSB (0) SRCSSB (1)
AMULTCOEFCMP	Use (1-coef) for texture coefficient
AMULTRJUSTIFY	Right shift 888 texture values to 555

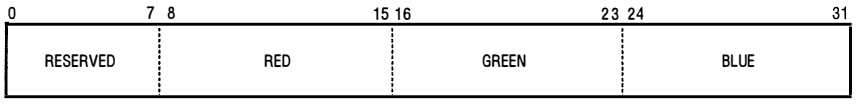


DBAMULTCONSTSSB0 - Texture Coefficient Constant 0 Register (0x0004_805c)

This register contains the texture RGB coefficient constant "0."

```
CLT_DBAMULTCONSTSSB) (pp, red, green, blue)
```

Field	Description
RED	Red constant 0
GREEN	Green constant 0
BLUE	Blue constant 0

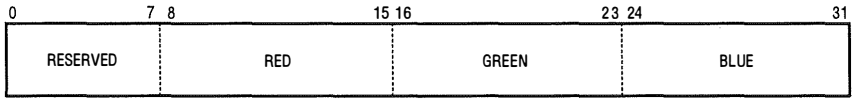


DBAMULTCONSTSSB1 - Texture Coefficient Constant 1 Register (0x0004_8060)

This register contains the texture RGB coefficient constant "1."

```
CLT_DBAMULTCONSTSSB1 (pp, red, green, blue)
```

Field	Description
RED	Red constant 1
GREEN	Green constant 1
BLUE	Blue constant 1



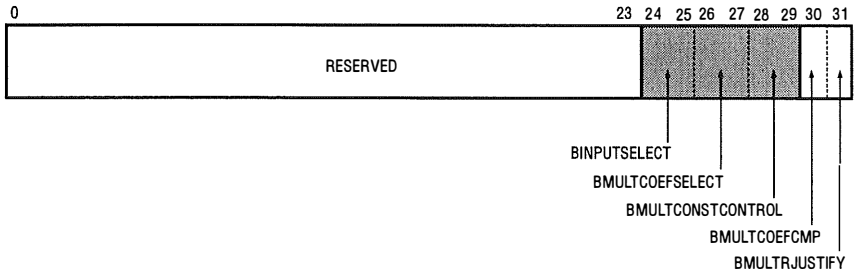
DBBMULTCNTL - Source Multiplication Control Register (0x0004_8064)

This register provides the source multiplication control.

```
CLT_DBBMULTCNTL (pp, binputselect, bmultcoefselect,
                 bmultconstcontrol, bmultrjustify)
```

Field	Description
BINPUTSELECT	Choose source or constant for source input: SRCCOLOR (0) CONSTCOLOR (1) TEXCOLORCOMPLEMENT (2) SRCALPHA (3)
BMULTCOEFSELECT	Choose source coefficient: TEXALPHA (00) SRCALPHA (08) CONST (10) TEXCOLOR (18) TEXALPHACOMPLEMENT (01) SRCALPHACOMPLEMENT (09) CONSTCOMPLEMENT (11) TEXCOLORCOMPLEMENT (19)
BMULTCONSTCONTROL	Choose constant controller: TEXSSB (0) SRCSSB (1)

Field	Description
BMULTCOEFCMP	Use (1-coef) for source coefficient
BMULTRJUSTIFY	Right shift 888 source values to 555

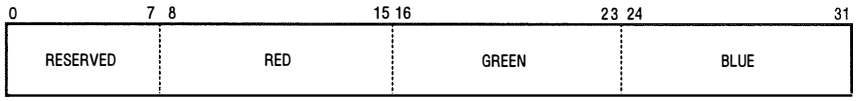


DBBMULTCONSTSSB0 - Source Coefficient Constant 0 Register (0x0004_8068)

This register contains the source RGB coefficient constant "0."

```
CLT_DBBMULTCONSTSSB0 (pp, red, green, blue)
```

Field	Description
RED	Red constant 0
GREEN	Green constant 0
BLUE	Blue constant 0

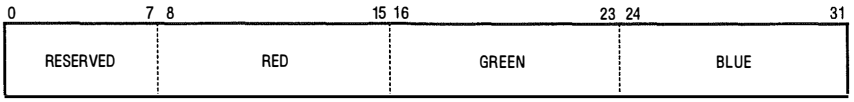


DBBMULTCONSTSSB1 - Source Coefficient Constant 1 Register (0x0004_806c)

This register contains the source RGB coefficient constant "1."

```
CLT_DBBMULTCONSTSSB1 (pp, red, green, blue)
```

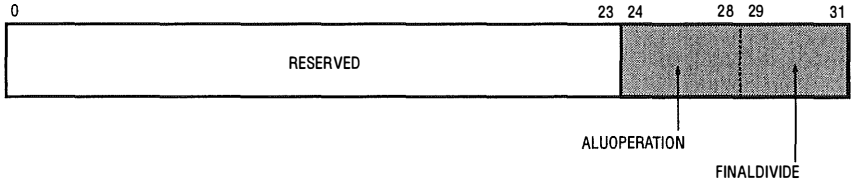
Field	Description
RED	Red constant 1
GREEN	Green constant 1
BLUE	Blue constant 1



DBALUCNTL - ALU Control Register (0x0004_8070)

```
CLT_DBALUCNTL (pp, aluoperation, finaldivide)
```

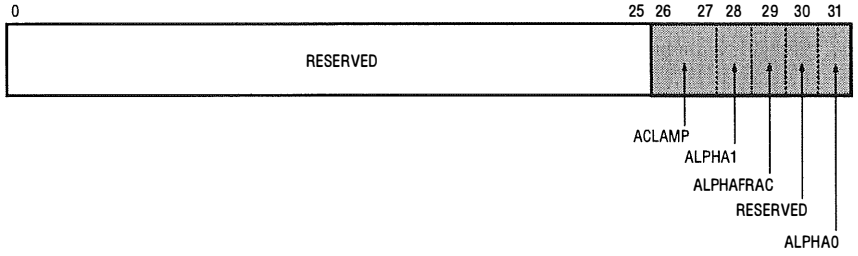
Field	Description
ALUOPERATION	ALU selection: A_PLUS_BCLAMP (0) A_PLUS_B (2) A_MINUS_BCLAMP (8) A_MINUS_B (a) B_MINUS_ACLAMP (c) B_MINUS_A (e) OUTPUTZERO (10) NEITHER (11) NOTA_AND_B (12) NOTA (13) NOTB_AND_A (14) NOTB (15) XOR (16) NOTA_AND_B (17) A_AND_B (18) ONEONEQUAL (19) B (1a) NOTA_OR_B (1b) A (1c) NOTB_OR_A (1d) A_OR_B (1e) OUTPUTONE (1f)
FINALDIVIDE	Final shift 0, 1, or 2



DBSRCALPHACNTL - Source Alpha Control Register (0x0004_8074)

```
CLT_DBSRCALPHACNTL (pp, aclamp, alpha1, alphafrac, alpha0)
```

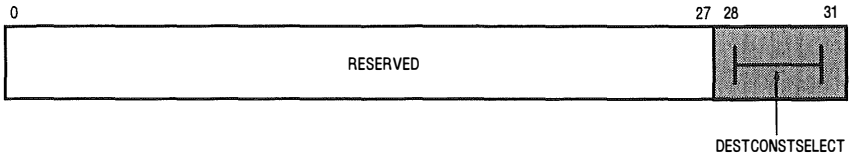
Field	Description
ACLAMP	Specify clamping option for three alpha ranges (3 @ 2 bit): LEAVEALONE (0) FORCE1 (1) FORCE0 (2)
ALPHA1	Two-bit field for Alpha equals 0xff
ALPHAFRAC	Two-bit field for fractional Alpha
ALPHA0	Two-bit field for Alpha equals 0



DBDESTALPHACNTL - Destination Alpha Control Register (0x0004_8078)

```
CLT_DBDESTALPHACNTL (pp, destconstselect)
```

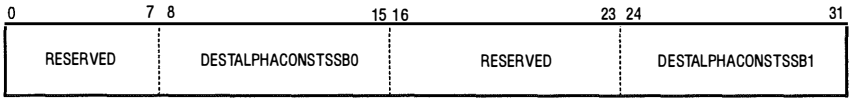
Field	Description
DESTCONSTSELECT	Combines meta select and select: TEXALPHA (0) TEXSSBCONST (1) SRCSSBCONST (9) SRCALPHA (2) RBLEND (3)



DBDESTALPHACONST - Destination Alpha Constants Register (0x0004_807c)

```
CLT_DBDESTALPHACONST (pp, destalphaconstssb0, destalphaconstssb1)
```

Field	Description
DESTALPHACONSTSSB0	Destination Alpha constant 0
DESTALPHACONSTSSB1	Destination Alpha constant 1



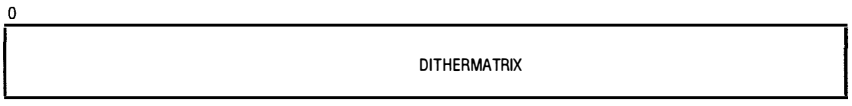
DBDITHERMATRIX - Dither Matrix Registers (0x0004_8080, 0x0004_8084)

This register contains the 4x4x4 dither matrix. The register at 0x0004_8080 holds the top two rows of the matrix, and is accessed using the `dmA` arguement.

The register at 0x0004_8084 holds the bottom two rows of the matrix, and is accessed using the `dmB` arguement.

```
CLT_DBDITHERMATRIX (pp, dmA, dmB)
```

Field	Description
X0Y0 - X3Y0/X0Y1 - X3Y1	First half of 4x4x4 dither matrix
X0Y2 - X3Y2/X0Y3 - X3Y3	Second half of 4x4x4 dither matrix

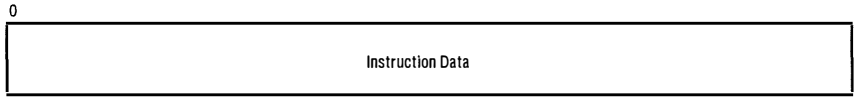


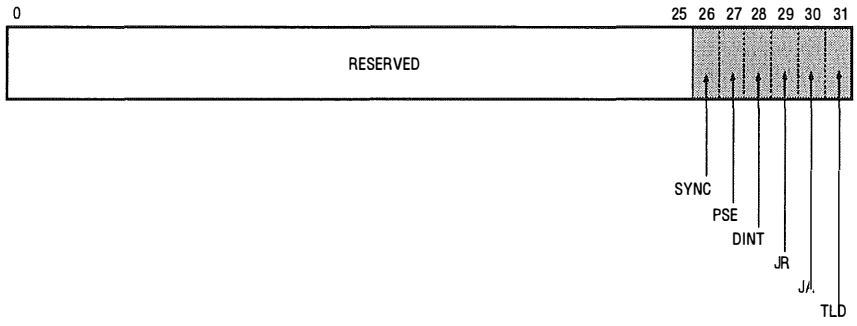
DCNTL - Deferred Flow Control Register (0x0004_0010, 0x0004_0014)

This register pair is used to perform various flow control instructions (see the "Flow Control Instructions" section, in Chapter 1, for more information). The data register is first loaded with the instruction data. The instruction register is then loaded with the instruction itself:

```
CLT_Sync (pp)
CLT_Pause (pp)
CLT_Interrupt (pp)
CLT_JumpRelative (pp)
CLT_JumpAbsolute (pp)
CLT_TxLoad (pp)
```

Field	Description
SYNC	Pause instruction execution until the entire TE pipe has been flushed
PSE	Pause execution
DINT	Interrupt CPU with vector - TEDCntlData is used as vector data
JR	Jump relative - use TEDCntlData as (2's complement) offset
JA	Jump absolute - use TEDCntlData as destination address
TLD	Texture load



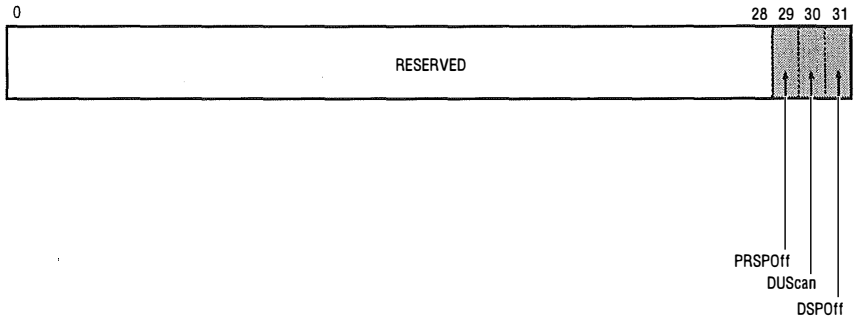


ESCNTL - Edge Walker/Span Walker Control Register (0x0004_4000)

This register contains three control bits. One turns perspective on and off, the second controls whether triangles are processed (scanned) from top-to-bottom or bottom-to-top, and the third turns DoubleStrike Prevention on and off.

```
CLA_ESCNTL (perspective, duscan, dspoff)
```

Field	Description
PRSPOff	Turn perspective calculations off.
DUScan	Down/up scan direction flags.
DSPOff	Disables Double Strike Prevention



Register Memory Map

Address	Name	Format	Access	Description
0x0004_8008	DBUserControl	Bit	RWSC	User Destination Blend Control
0x0004_800C	DBDiscard Control	Bit	RWSC	Pixel discard control
0x0004_8010	DBStatus	Bit	RWSC	Status indication
0x0004_8014	DBIntCntl	Bit	RWSC	Mask indicating which ALU and Z results should generate interrupts.
0x0004_801C	DBXWinClip	U_Fix	RW	X window clip values
0x0004_8020	DBYWinClip	U_Fix	RW	Y window clip values

Address	Name	Format	Access	Description
0x0004_8030	DBSrcCntl	Bit	RWSC	Source read Control register
0x0004_8034	DBSrcBaseAddr	Addr	RW	Source read base pointer
0x0004_8038	DBSrcXStride	U_Fix	RW	Source read X Stride value - Distance in pixels to next line.
0x0004_803C	DBSrcOffset	S_Fix	RW	X and Y Offset used for Source reads
0x0004_8040	DBZCntl	Bit	RWSC	Z Buffer Control register
0x0004_8048	DBZOffset	S_Fix	RW	X and Y Offset used for Z.
0x0004_8050	DBSSBDSBCntl	Bit	RWSC	SSB & DSB Control register
0x0004_8054	DBConstIn	U_Fix	RW	Const RGB used for computation input
0x0004_8058	DBAMultCntl	Bit	RWSC	Texture color multiplication control
0x0004_805C	DBAMultConst SSB0	U_Fix	RW	Texture RGB coef const "0"
0x0004_8060	DBAMultConst SSB1	U_Fix	RW	Texture RGB coef const "1"
0x0004_8064	DBBMultCntl	Bit	RWSC	Texture color multiplication control
0x0004_8068	DBBMultConst SSB0	U_Fix	RW	Source RGB consts "0"
0x0004_806C	DBBMultConst SSB1	U_Fix	RW	Source RGB consts "1"
0x0004_8070	DBALUCntl	Bit	RWSC	ALU Control register
0x0004_8074	DBSrcAlphaCntl	Bit	RWSC	Control of source alpha
0x0004_8078	DBDest AlphaCntl	Bit	RWSC	Control of destination alpha
0x0004_807C	DBDestAlpha-Const	U_Fix	RW	Destination Alpha constants

Address	Name	Format	Access	Description
0x0004_8080	DBDither MatrixA	Bit	RWSC	First half of 4x4x4 dither matrix
0x0004_8084	DBDither MatrixB	Bit	RWSC	Second half of 4x4x4 dither matrix
0x0004_808C-0x0004_9FFC	reserved		NA	

Note: *Registers are not initialized at reset unless stated otherwise. They must be explicitly set by the OS or user. Once a register has been set, it will retain its value throughout all rendering until it is altered by a list instruction or the CPU directly. Note that the triangle engine must go through a sync operation when a control register is changed. Changing registers "on the fly" will produce unpredictable results and in some circumstances may hang the triangle engine until it is reset.*

Appendix A

Register Setups

This appendix lists the Triangle Engine register configurations for the following operations:

- Z-buffering
- Dithering
- Transparencies
- Texturing (on/off)
- Perspective (on/off)
- Tiling a texture
- Loading an uncompressed texture
- Loading a PIP table

Z-Buffering

Turn Z-buffering on:

```
CLT_SetRegister(pp,DBUSERCONTROL,CLA_DBUSERCONTROL(1,1,0,0,0,0,0,0)
)
```

Select which pixels are removed by the Z-buffer (only pixels that are closer than what is in the Z-buffer will be drawn):

```
CLT_DBZCNTL(ptr, 0,0,0,0,1,1)
```

Set the Z offset:

```
CLT_DBZOFFSET(ptr,0,0)
```

You will be unable to do Z-buffering unless you've allocated a Z-buffer and told the TE about it (typically with GS_SetZBuffer). You will also have to clear your Z-buffer every frame, and include perspective values (w) with your triangles.

Dithering

Turn dithering on:

```
CLT_SetRegister(pp,DBUSERCONTROL,CLA_DBUSERCONTROL(0,0,0,0,0,0,1,0
))
```

Set the dithering matrix to a basic dither pattern (see Figure A-1):

```
CLT_DbDitherMatrix(ptr,0xC0D12E3F,0xE1D0302F)
```

$$\begin{bmatrix} -4 & 0 & -3 & 1 \\ 2 & -2 & 3 & -1 \\ -2 & 1 & -3 & 0 \end{bmatrix}$$

Figure E-1 *Dithering matrix*

Setting Up the Destination Blender Source Buffer

To set up the frame buffer as a secondary source, set the following registers:

DBUSERCONTROL can be used to turn blending on, and to enable blending with a source frame buffer.

DBSRCBASEADDR points to the address of the frame buffer bitmap

DBSRCCNTL specifies whether the frame buffer is 16 or 32 bits

DBSRCOFFSET specifies an X and Y offset to use before reading from the frame buffer. This should usually be set to (0,0)

DBSRCXSTRIDE sets the width of the source frame buffer

Transparencies

There are a number of ways to do a transparency:

•To cause black (RGB 0,0,0) pixels, pixels with SSB 0, or pixels with Alpha 0 to be totally transparent, simply use the DBDISCARDCONTROL register. (Note: this checking takes place after all texturing has been done.)

•To set a constant amount of transparency (for instance, 90%), first set up the frame buffer as a secondary source (see above). Then call:

CLT_DBAMULTCNTL(ptr, RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR, RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST, 0, 0) to set the first source to come from the triangle data

CLT_DBBMULTCNTL(ptr, RC_DBBMULTCNTL_BINPUTSELECT_SRCCOLOR, RC_DBBMULTCNTL_BMULTCOEFSELECT_CONST, 0, 0) to set the second source to come from the source data

CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0) to set the final math to add the two sources together

CLT_DBAMULTCONSTSSB0(ptr, 255-TRANS, 255-TRANS, 255-TRANS)

CLT_DBAMULTCONSTSSB1(ptr, 255-TRANS, 255-TRANS, 255-TRANS)

CLT_DBBMULTCONSTSSB0(ptr, TRANS, TRANS, TRANS)

CLT_DBBMULTCONSTSSB1(ptr, TRANS, TRANS, TRANS) to set up the degree of transparency, where TRANS is between 0 and 255, with 0 meaning totally opaque and 255 meaning totally transparent

•To set transparency to depend on the Alpha channel, first set up the frame buffer as a secondary source (see above). Then call:

CLT_DBAMULTCNTL(ptr,
RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
RC_DBAMULTCNTL_AMULTCOEFSELECT_TEXALPHA, 0, 0);

CLT_DBBMULTCNTL(ptr, RC_DBBMULTCNTL_BINPUTSELECT_SRCCOLOR,
RC_DBBMULTCNTL_BMULTCOEFSELECT_TEXALPHACOMPLEMENT, 0, 0);

CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0)

Then an alpha value of 1.0 will result in a completely opaque triangle, while an alpha of 0.0 will result in a completely transparent triangle, etc.

Texturing

To turn texturing on and off, use the lookup enable bit of the TXTADDRCNTL register. If you leave this bit set, you may get a performance hit even if you are drawing untextured triangles. However, there are a number of other registers that must be set before texturing can be used (not to mention that a texture load must be done before a texture can be drawn):

TXTADDRCNTL must also be used to select a filtering mode and set the number of LODs present.

TXTPIPCNTL must be set, even if you're using a literal (unindexed) texture. For a literal texture, it should be set so that the SSB, alpha and color come from the texture cache. For an indexed texture, the color should presumably come from the PIP, and the alpha and SSB can come from wherever you like. For indexed textures, this register also contains the PIP index offset, allowing you to select which part of the PIP will be used.

TXTEXPTYPE must be set. This specifies what type of texture is being used (how many bits of color, whether it is indexed or unindexed, etc.) This will typically be set during your texture load

TXTUVMASK must be set. This register is used to tile textures (see below), but for untiled textures it should be set to (0x3ff, 0x3ff)

TXTUVMAX must be set. This sets the size of the texture, and will typically be set during your texture load. If multiple LODs are present, this register refers to the size of the smallest, and thus coarsest, LOD. Note that the values contained in this register are the dimensions of the texture, in pixels, minus one.

TXTLODBASE[0-(n-1)] must be set, where n is the number of LODs present. These registers specify the starting locations of the texture LODs in TRAM. They will usually be set during the texture load.

TXTTABCNTL must be set up properly. If texturing is turned off, then there will be only two inputs to the texture blender (PRIMCOLOR and PRIMALPHA) so any blends used can only have them as inputs. Typically, you will just want to set COLOROUT to PRIMCOLOR and ALPHAOUT to PRIMALPHA. If texturing is

turned on, then there are many many more options. However, if you actually want your texture to show up, you'll need to set COLOROUT to either TEXCOLOR or some BLEND one of whose inputs is TEXCOLOR.

ESCNTL must be used to turn perspective correction either on or off, according to your needs. If it is set incorrectly, your textures will be warped and unviewable.

Perspective

Turn perspective correction on:

```
CLT_SetRegister(pp, ESCNTL, CLA_ESCNTL (1,0,0))
```

Turn perspective correction off:

```
CLT_ClearRegister(pp, ESCNTL, CLA_ESCNTL (1,0,0))
```

Remember that if perspective correction is on, the texture coordinates entered in vertex commands must be multiplied by 1/w, but if perspective correction is off, raw texture coordinates must be used.

Sprites

To draw sprites, that is, to use the TE to simply draw textures onto the screen very quickly, make sure that blending is off, secondary source is off, perspective correction is off, Z-buffering is off (unless it's needed), and point filtering is being used. Be aware, however, that if you just want to copy large rectangular regions of data with the TE, it's faster to set a secondary source to point to the source data, enable source blending in the destination blender, and use a strip or fan of two untextured triangles to copy the data. However, this will only work if the source data is in literal 16- or 32-bit format, and if no special effects (such as black regions being transparent) are needed.

Lighting

To do lighting, you'll first need a 3D lighting model, which is a very complicated issue in its own right. However, there are a few typical things that might be done with the CLT to implement the graphical side of the lighting model. In particular, a lighting model will frequently be used with textured triangles, and it will want to do two things, namely darken the texture where the texture is not illuminated, and brighten the texture where there should be specular highlights.

To do either of these things on a triangle-by-triangle basis requires that there be color data for the vertices, but that color data can be used several different ways.

Lighting method #1: With this method, the RGB values of the color data are used to select the color of a light, and the alpha value sets how strong that light is. More specifically, the texture application blender is used, with the iterated alpha controlling the LERP between the iterated color and the textured color. So, to darken the texture, you can specify an RGB color of black. Then an alpha of 1 would result in black, an alpha of 0.5 would result in the texture being displayed at 50% intensity, an alpha of 0 would result in the normal texture, and so on. To brighten the texture, you can specify on RGB of white, or whatever color you want your specular highlights to be. The advantage of this method is that it uses only the texture application blender, so the destination blender is free to produce other effects, or to be turned off to save performance. Furthermore, this method allows you to still use your textured alpha for transparency or whatever, even though your iterated alpha is being taken up in the lighting. The disadvantage is that it assumes

that you've combined all of your lighting calculations and arrived at one final output color, instead of keeping the darkening and brightening effects separate. To use this method, simply set up the texture application blending as follows:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMCOLOR,
RC_TXTTABCNTL_THIRDCOLOR_PRIMALPHA, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_TEXALPHA,
RC_TXTTABCNTL_BLENDOP_LERP);
```

Lighting method #2: With this method, the RGB color is used to darken the texture, and the alpha value is used to brighten the color. In this case, both the texture and destination blenders are used, with the darkening taking place in the texture blender and the lightening taking place in the destination blender. With this method, the RGB values are used as multipliers for the texture colors, so RGB values of (0.5, 0.5, 0.5) will result in the texture being displayed at half intensity. However, the alpha value is used to brighten the color, after the darkening has taken place, with an alpha of 0 not affecting the texture at all, and an alpha of 1 brightening all the way to solid white. This method is easier to use with most lighting models that treat specular highlights separately from the other lighting, but it uses both the texture and destination blenders, and doesn't allow the texture alpha to be accessed at all. To use this method, set up the blending as follows:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMCOLOR,
RC_TXTTABCNTL_THIRDCOLOR_PRIMALPHA, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_PRIMALPHA,
RC_TXTTABCNTL_BLENDOP_MULT);

CLT_DBAMULTCNTL(ptr,
RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST, 0, 0);

CLT_DBAMULTCONSTSSB0(GS_Ptr(gs), 255, 255, 255);

CLT_DBAMULTCONSTSSB1(GS_Ptr(gs), 255, 255, 255);

CLT_DBBMULTCNTL(ptr,
RC_DBBMULTCNTL_BINPUTSELECT_TEXCOLORCOMPLEMENT,
RC_DBBMULTCNTL_BMULTCOEFSELECT_TEXALPHA, 0, 0);

CLT_DBALUCNTL(ptr, RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0)
```

Simpler lighting models: If your lighting model doesn't require specular highlights, it is easy to simply have the RGB color or the alpha value multiply the texture value. This is accomplished with the texture application blender. For instance, to have the alpha value multiply the texture value, call:

```
CLT_TXTTABCNTL(ptr, RC_TXTTABCNTL_FIRSTCOLOR_TEXCOLOR,
RC_TXTTABCNTL_SECONDCOLOR_PRIMALPHA, 0, 0, 0,
RC_TXTTABCNTL_COLOROUT_BLEND,
RC_TXTTABCNTL_ALPHAOUT_TEXALPHA,
RC_TXTTABCNTL_BLENDOP_MULT);
```

Tiling a Texture

First of all, you can only tile a texture if its width or height (depending on which direction you're tiling in) is a power of two. To turn on tiling, you'll need to set the TXTUVMASK and TXTUVMAX registers in a fairly trick fashion. Each of these registers has both an X- and a Y-component. To tile your texture in the X direction, first set the X component of TXTUVMASK to txwidthmax-1, where txwidthmax is equal to the width, in pixels, of the largest LOD. Then set the X component of TXTUVMAX to 0x3ff >> (numLOD-1), where numLOD is the number of LODs present in the texture. To turn off tiling in the X direction, set the X component of TXTUVMASK to 0x3ff and the X component of TXTUVMAX to txwidth-1, where txwidth is the width, in pixels, of the smallest (coarsest) LOD.

Similarly, to tile your texture in the Y direction, set the Y component of TXTUVMASK to txheightmax-1 and the Y component of TXTUVMAX to 0x3ff>>(numLOD-1). To turn tiling off in the Y direction, set the Y component of TXTUVMASK to 0x3ff and the Y component of TXTUVMAX to txheight-1.

Load an Uncompressed Texture

To do a texture load of an uncompressed texture, make the following calls:

```
CLT_TXTLDCNTL(ptr, 0, RC_TXTLDCNTL_LOADMODE_TEXTURE, bitoffset)
```
Where bitoffset is the offset into the byte in main memory of the start of the texture to be loaded, in bits.

```
CLT_TXTCOUNT(ptr, textureheight)
```
Where textureheight is the number of rows in the texture.

```
CLT_TXTLDSRCADDR(ptr, txaddress)
```
Where txaddress is the address in memory of the texture to be loaded.

```
CLT_TXTLODBASE0(ptr, tramaddress)
```
Where tramaddress is the word-aligned address in TRAM where the texture will be placed (ranging from 0 to 0x3fff). If you're only going to have one texture in TRAM at a time, this should probably be 0.

```
CLT_TXTLDWIDTH(ptr, txwidth, txsourcewidth)
```
Where txwidth is the width of the texture, in bits, and txsourcewidth is the width of the region from which the texture is being loaded, in bits.

```
CLT_TxLoad(ptr)
```
Load the texture.

Loading a PIP Table

To perform a PIP load, make the following calls:

```
CLT_TXTLODBASE0(ptr, 0x46000)
```
Tells the texture loader to load the PIP into PIP RAM.

```
CLT_TXTLDCNTL(ptr, 0, RC_TXTLDCNTL_LOADMODE_PIP, 0)
```

```
CLT_TXTLDSRCADDR(ptr, pipsourceaddress)
```
Where pipsourceaddress is the word-aligned address in main memory where the PIP is located.

```
CLT_TXTCOUNT(ptr, pipsize)
```
Where pipsize is the size, in bytes, of the PIP to load. A full 256 entry PIP takes up 1024 bytes.

```
CLT_TxLoad(ptr)
```

Appendix B

Sample Code

This is a sample program that uses Gstate and the Command List Toolkit. It draws random triangles up on the screen very rapidly.

```
/*All of the following are nice basic include files*/
#include <stdio.h>
#include <stdlib.h>
#include <kernel:types.h>
#include <graphics:clt:clt.h>
#include <graphics:clt:gstate.h>
#include <graphics:graphics.h>
#include <graphics:view.h>
#include <graphics:error.h>
#include <kernel:io.h>
#include <kernel:mem.h>
#include <kernel:random.h>
#include <assert.h>

#define TRISPERLIST 100

/*This constant specifies how many triangles each command list will draw. If
 *we wanted to we could have each command list we send to the TE have only
 *one triangle in it, but there is some overhead to sending lists, so it's
 *best to have each list not be tiny.
 */

int32 main(void)
{

   GState *gs;
       /*the heart of using GState is creating a GState struct... */

   Item   viewItem; /*we need a View so that things will show up on screen*/
```

```
   int i;

   Item bitmaps[1];
    /*we need a bitmap item for the view to display,
     *and for the TE to render into
     */

OpenGraphicsFolio (); /*must always be called before doing graphics...*/

gs=GS_Create();
    /*This call creates and initializes the GState*/

GS_AllocLists(gs, 2, 4096);
    /*This call allocates the command lists for the GState. In this case,
     *we'll have two command lists, each of 4096 words
     */

GS_AllocBitmaps( bitmaps, 320, 240, BMTYPE_16, 1, 0);
    /*this call allocates a bitmap for us*/

   GS_SetDestBuffer(gs, bitmaps[0]);
    /*this call tells the GState which bitmap to render into*/

viewItem = CreateItemVA(MKNODEID(NST_GRAPHICS, GFX_VIEW_NODE),
                  VIEWTAG_VIEWTYPE, BMTYPE_16,
                  /*specifies 320x240x16bit*/

                  VIEWTAG_BITMAP, bitmaps[0],
                  /*Point the view towards our first bitmap*/

                  TAG_END );
    /*this call creates the View Item, which is necessary for
     *things to show up on screen
     */


AddViewToViewList( viewItem, 0 );
    /*This causes the view to actually be displayed*/

    /*the next several commands are all CLT commands, all of
     *which are register commands. For instance, the first
     *command is CLT_DBUSERCONTROL. It's passed a pointer to
     *a command list, and a bunch of numbers. What it does is
     *it adds a command to the command list. That command
     *sets the state of the DBUSERCONTROL register, which is
     *nice, basic TE register. The value it sets DBUSERCONTROL
     *to comes from the arguments. In the case of DBUSERCONTROL,
     *most of these arguments represent a single flag bit.
     */

    /*note that the moral of this story is that before _anything_
     *will show up on the screen, there are quite a few registers
     *that need to be set up, but once they're all set up, they
     *can be more or less ignored
     */
```

```
   CLT_DBUSERCONTROL(GS_Ptr(gs), 0,0,0,0,1,1,0,15);
       /*basic initialization. In this case, Z-buffering is off,
        *window clipping is off, destination blending is on, source
        *blending is on, and diterhing is off
      */


       /*this next command also is a register setting command,
        *in this case setting the DBDISCARDCONTROL register
        */
      CLT_DBDISCARDCONTROL(GS_Ptr(gs),0,0,0,0);
       /*all pixel discards are turned off. these would be turned on, for
       *instance, to make all black pixels transparent
        */

      CLT_DBCONSTIN(GS_Ptr(gs), 0,0,0);
       /*set the constant color used in destination blending to black
         *(destination blending takes the color and texture from the triangle
      *and does a number of odd things to it. In this case, we'll just add
      *it to black, which effectively leaves it unchanged)
      */

      CLT_DBAMULTCONSTSSB0(GS_Ptr(gs), 0xff, 0xff, 0xff);
      CLT_DBAMULTCONSTSSB1(GS_Ptr(gs), 0xff, 0xff, 0xff);
      CLT_DBBMULTCONSTSSB0(GS_Ptr(gs), 0xff, 0xff, 0xff);
      CLT_DBBMULTCONSTSSB1(GS_Ptr(gs), 0xff, 0xff, 0xff);
       /*set all of our multiplying constants for the destination blender
         *to 1, so we can just pass colors directly through
      */

      CLT_DBALUCNTL(GS_Ptr(gs), RC_DBALUCNTL_ALUOPERATION_A_PLUS_B, 0);

     /*This set the final ALU stage of the destination blending to just add the
      *A and B colors together. In this case, the A color will be the
      *shading from the triangle we're drawing and the B color will be black
       */

       CLT_TXTADDRCNTL(GS_Ptr(gs), 0, 0, 0, 0, 0);
     /*Turn texturing off*/

       CLT_DBAMULTCNTL(GS_Ptr(gs), RC_DBAMULTCNTL_AINPUTSELECT_TEXCOLOR,
        RC_DBAMULTCNTL_AMULTCOEFSELECT_CONST,
RC_DBAMULTCNTL_AMULTCONSTCONTROL_TEXSSB, 0);

/*sets the A input to the destination blender to be the color from the texture
*mapper. Note that even though we aren't doing any texturing we want the color
*from the texture mapper, because the texture mapper handles all shading.
*/

      CLT_DBBMULTCNTL(GS_Ptr(gs), RC_DBBMULTCNTL_BINPUTSELECT_CONSTCOLOR,
        RC_DBBMULTCNTL_BMULTCOEFSELECT_CONST,
        RC_DBBMULTCNTL_BMULTCONSTCONTROL_TEXSSB, 0);

      /*sets the B input to the destination blender to just be a constant color
```

```
      *(which in this case we earlier set to black)
      */

     GS_SendList(gs);

/*This command is totally different from all the CLT macros we've been using.
*It tells the GState to take the command list it's been assembling, which
*consists of a bunch of register instructions, and send it to the TE.
*Note that whenever you call SendList the Gstate will stick a CLT_Pause
*command at the end of your command list, which means that you don't have
*to remember to do so yourself. But if you're ever sending lists
*yourself, always remember to terminate them with a CLT_Pause command.
*/

   while(1)
      {
      for (i=0;i<TRISPERLIST;i++)
       {

        CLT_TRIANGLE(GS_Ptr(gs), 1, RC_STRIP, 0, 0, 1, 3);
          /*To set up a command list to draw a triangle, we first need a
           *triangle command. The second argument says that this is a new
           *tristrip/fan (as opposed to connecting to a previously drawn one.
           *The third argument says that it's a strip, not a fan.
           *The fourth argument says that it has no perspective info (z
           *values). The fifth argument says that it has no texturing.
           *The sixth argument says that it does have RGBA color info.
           *The final argument says that it has 3 vertices.
           *
           *After the triangle command we'll need to issue one vertex command
           * for each vertex of the triangle. Note that there can not be any
           *other TE instructions between the triangle command and the vertex
           *commands. Also, if you give the wrong number of vertex commands,
           *or if you, say, include
          *perspective info in the vertex command but don't mention it in the
          *triangle command, then bad things will happen. Namely, the TE will
           *probably time out.
           */


        CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
            (gfloat)(rand() % 10000)/10000.0,
         (gfloat) (rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/10000.0,0);
        CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
            (gfloat)(rand() % 10000)/10000.0,
         (gfloat) (rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/10000.0,0);
        CLT_VertexRgba(GS_Ptr(gs), (gfloat)(rand()%320), (gfloat)(rand()%240),
            (gfloat)(rand() % 10000)/10000.0,
          (gfloat) (rand() % 10000)/10000.0,(gfloat)(rand() % 10000)/
10000.0,0);

    /*CLT_VertexRgba is the macro for vertices with shading but no perspective
    *or texture. Since perspective, texture, and shading can all be turned
    *on or off independently, there are a total of 8 possible distinct
    *types of vertex command, each with an accompanying macro.
           */
```

```
        CLT_Sync(GS_Ptr(gs));

    /*after drawing a triangle, but before setting any registers, you should
    *always call CLT_Sync, which inserts a sync command into the command list.
     *In this particular case, we don't set any registers after drawing the
     *triangle, so we don't really need this sync command, but I needed an
     *excuse to mention it, since it's so important. Get in the habit of
     *sticking it after triangle commands.
           */
      }

     GS_SendList(gs);
      /*This sends our newly created command list with instructions for
       * drawing 100 triangles to the Triangle Engine
       */

     }
   CloseGraphicsFolio();
   /*and we're done!*/

 }
/*have a nice day*/
```

Appendix C

Function Calls

This Appendix lists all of the Command List Toolkit function calls.

GState

GS_AllocBitmaps .....Utility to allocate frame buffers and Z-buffer
GS_AllocLists .....Allocate one or more command list buffers for graphics rendering
GS_BeginFrame.....Notifies a GState that the next cmd. list is the first for a frame
GS_Create.....Creates a GState (Graphics State) object
GS_Delete.....Frees all memory associated with a GState object
GS_FreeBitmaps .....Utility to free bitmaps and Z-buffer
GS_FreeLists .....Free the memory used by command lists for a GState object
GS_GetCmdListIndex.....Returns index of which cmd list buffer is in use
GS_GetCount.....Get the current GState counter
GS_GetCurListStart .....Get ptr to start of the current command list buffer
GS_GetDestBuffer.....Get the Item number of the bitmap being used as an output frame buffer
GS_GetVidSignal .....Returns the signal bit which a GState is using for video synchronization
GS_GetZBuffer.....Get the Item number of the bitmap being used as a Z-buffer
GS_Init OBSOLETE .....Initializes a GState object to a default state
GS_Reserve .....Reserve block of memory in GState's current command list buffer
GS_SendList.....Send a GState's current command list to the Triangle Engine
GS_SetDestBuffer.....Set a bitmap as the output frame buffer for a GState
GS_SetList.....Set a GState to use a new command list buffer
GS_SetVidSignal.....Attach a signal to a GState object, for video synchronization
GS_SetZBuffer.....Set a bitmap as the Z-buffer for a GState
GS_WaitIO .....Wait for all pending IO to complete for a GState

Triangle Engine Command List Toolkit

CLT_AllocSnippet ..... Allocate space for the data for a CltSnippet
CLT_ClearFrameBuffer..... Clear frame buffer and/or Z-buffer via CLT cmds
CLT_ComputePipLoadCmdListSize ..... Compute size of PIP load command list
CLT_ComputeTxLoadCmdListSize..... Compute size of texture load command list
CLT_CopySnippetData..... Copy data portion of a cmd list snippet to specified location
CLT_CreatePipCommandList..... Creates a cmd list snippet to load a PIP
CLT_CreateTxLoadCommandList ..... Creates a cmd list snippet to load a texture
CLT_FreeSnippet ..... Free memory allocated for a CltSnippet's data
CLT_InitSnippet..... Initializes a CltSnippet structure
CLT_SetSrcToCurrentDest..... Set the Dest Blender so that src blends will occur with cur frame buffer

Globals

CltEnableTextureSnippet..... Global var. used to enable texturing
CltNoTextureSnippet..... Global var. used to disable texturing

GS_AllocBitmaps

Utility to allocate frame buffers and Z-buffer

Synopsis

Err GS_AllocBitmaps(Item bitmaps[], uint32 xres, uint32 yres,
uint32 bmType, uint32 numFrameBufs, bool useZb);

Description

Utility routine to allocate frame buffers, and, optionally, also a Z-buffer. Since the Triangle Engine is able to perform better when the Z-buffer is aligned to an odd page offset from the frame buffer(s), **GS_AllocBitmaps()** will handle aligning the buffers automatically.

Arguments

bitmaps
: Array of Items which will become the bitmaps. This array should be large enough to contain one item per frame buffer, plus one more if Z-buffering is going to be used. Upon successful return, bitmaps will contain the specified number of frame buffers in bitmaps[0] ... bitmaps[numFrameBufs-1]. If a Z-buffer is needed, it will be allocated in bitmaps[numFrameBufs].

xres
: The width, in pixels, of each bitmap

yres
: The height, in pixels, of each bitmap

bmType
: Specifies the type of bitmaps to be created. This value should be one of the constants defined in the *<:graphics:bitmap.h>* header file, such as BMTYPE_32 or BMTYPE_16.

numFrameBufs
: The number of frame buffers to be allocated. This number should NOT include the Z-buffer

useZb
: Specifies whether a Z-buffer should be allocated. If TRUE, then the size of the bitmaps[] array should be numFrameBufs+1.

Return Value

Returns >= 0 for success or a negative error code for failure

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_FreeBitmaps()**, **GS_SetDestBuffer()**, **GS_SetZBuffer()**

GS_AllocLists

Allocate one or more command list buffers for graphics rendering

Synopsis

Err GS_AllocLists(GState* g, uint32 nLists, uint32 listSize);

Description

Allocates one or more command lists buffers of the specified size. These buffers are used by the Font Folio, 2D Graphics Folio, and 3D Graphics Library to build command lists for the Triangle Engine to render.

Each command list buffer must be large enough that the largest triangle strip or fan in a program's data can fit within the buffer. Triangle strips and fans can sometimes be several hundred vertices long, and each vertex can take up to 9 32-bit words (36 bytes) of command list buffer space.

Normally, a minimum of two command list buffers should be allocted, so that while the Triangle Engine is rendering one buffer's commands, the CPU can begin preparing the next set of rendering commands. Larger buffers will often yield better performance than smaller buffers, since the CPU spends less time flushing buffers and waiting for new ones to be freed up by the Triangle Engine. The parameters to **GS_AllocLists()** should be adjusted during development of a title, to optimally blend between high performance and memory usage.

Arguments

g
: Pointer to a GState object

nLists
: Number of command lists buffers to allocte.

listSize
: Size of each list to be allocated, in WORDS.

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_FreeLists()**, **GS_Reserve()**, **GS_SendList()**, **GS_SetList()**

GS_BeginFrame

Notifies a GState that the next cmd. list is the first for a frame

Synopsis

Err GS_BeginFrame(GState* g);

Description

Notifies a GState that the next command list buffer is the first to be rendered to a Bitmap. This routine is used when double- buffering is desired. At the first call to **GS_SendList()** after **GS_BeginFrame()** is called, the GState will wait on a signal from the Display Folio before sending a command list to the Triangle Engine device driver.

In order for double-buffering to correctly prevent screen tearing, this routine should be called after each call to switch between frame buffers (usually with a call to **ModifyGraphicsItem()**). In addition, a signal should be associated with the View and a GState when these objects are being created. See **GS_SetVidSignal()** for more information about how to correctly allocate this signal.

Arguments

g
: Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SetVidSignal()**, **GS_GetVidSignal()**, **GS_SendList()**

 **GS_AllocLists**  **GS_Create**

GS_Create

Creates a GState (Graphics State) object

Synopsis

GState* GS_Create(void);

Description

Creates a GState, or Graphics State, object. This object is used to encapsulate the information which needs to be maintained for command list buffers. The Triangle Engine operates by executing the series of instructions which are placed into these command list buffers.

The 2D graphics folio, font folio, and 3D graphics library all use command list buffers to communicate with the Triangle Engine. By creating a common GState that can be shared by these folios and libraries, the user can combine text, 2D graphics, and 3D graphics more easily.

After a GState is created, it still must be intialized by calling **GS_AllocLists()**, to create one or more buffers of the specified size.

Return Value

Returns a pointer to a GState if successful, or NULL if it fails.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_AllocLists(), **GS_Reserve()**, **GS_SendList()**, **GS_SetList()**, **GS_WaitIO()**

GS_Delete

Frees all memory associated with a GState object

Synopsis

Err GS_Delete(GState* g);

Description

Frees up all memory used by a GState object. If command list buffers have been allocated by calling **GS_AllocLists()**, **GS_Delete()** will also free the memory used by these buffers.

Arguments

g
Pointer to a GState object

Return Value

Returns 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_FreeLists()**

 **GS_Create**  **GS_FreeBitmaps**

GS_FreeBitmaps

Utility to free bitmaps and Z-buffer

Synopsis

Err GS_FreeBitmaps(Item bitmaps[], uint32 numBitmaps);

Description

Free all bitmaps in the bitmaps[] array. Usually, these will be bitmaps that were created by calling **GS_AllocBitmaps()**.

Note that if any of these bitmaps were in use by a GState object, an error will be returned the next time that GState needs to use one of the bitmaps. Therefore, it is a good idea to always either assign new bitmaps to a GState before freeing its bitmaps, or to free a GState, by calling **GS_Delete()**, before freeing the array of bitmaps.

Arguments

bitmaps
: Array of items representing the bitmaps for the frame buffer(s), and optionally, a Z-buffer.

numBitmaps
: The total number of frame buffers in the bitmaps[] array. This number should equal the number of frame buffers allocated, plus one if a Z-buffer was used.

Return Value

Returns >= for success or a negative error code for failure

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_Delete()**, **GS_AllocBitmaps()**

GS_FreeLists

Free the memory used by command lists for a GState object

Synopsis

Err GS_FreeLists(GState* g);

Description

Free the memory used by the command list buffer(s) for a GState object. These buffers should have been allocated by calling **GS_AllocLists()**.

Normally, this routine does not need to be called, since it will be called by **GS_Delete()** if the command list buffers are still allocated. However, it can be used in special situations, such as when, because of memory considerations at run-time, it becomes necessary to re-size the command list buffers.

Arguments

g
: Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_Delete()**

 **GS_FreeBitmaps**  **GS_GetCmdListIndex**

GS_GetCmdListIndex

Returns index of which cmd list buffer is in use

Synopsis

uint32 GS_GetCmdListIndex(GState* g)

Description

This routine returns the index into the command list buffer array of the command list buffer which is currently in use. In most cases, this routine will only be useful to people writing their own routine to send a command list to the Triangle Engine.

In the case where more than one command list buffer is allocated when **GS_AllocLists()** is called, each call to **GS_SendList()** will increment this index so that the CPU can begin filling the next command list buffer.

Arguments

g
: Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_SendList()**

GS_GetCount

Get the current GState counter.

Synopsis

uint32 GS_GetCount(GState *gs);

Description

Returns the number of times that **GS_WaitIO()** has completed.

Arguments

g

Pointer to a GState object

Return Value

Returns the count value. If the gs parameter is invalid, then the return value will be garbage.

Implementation

DLL call implemented in gstate V30

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

 **GS_GetCmdListIndex**  **GS_GetCurListStart**

GS_GetCurListStart

Get ptr to start of the current command list buffer

Synopsis

CmdListP GS_GetCurListStart(GState* g)

Description

Returns a pointer to the start of the current command list buffer. In most cases, this routine will only be useful to people writing their own routine to send a command list to the Triangle Engine.

Arguments

g

Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SendList()**

GS_GetDestBuffer

Get the Item number of the bitmap being used as an output frame buffer

Synopsis

Item GS_GetDestBuffer(GState* g);

Description

Returns the Item number for the Bitmap being used as an output frame buffer for a GState object. When Triangle Engine commands are rendered through the specified GState object, the Triangle Engine will render into this Bitmap. Z-buffer information is stored in a separate bitmap.

Arguments

g
Pointer to a GState object

Return Value

Returns an Item number for the Bitmap, or an error code if a valid bitmap has not been set for this GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SetDestBuffer()**, **GS_SetZBuffer()**, **GS_GetZBuffer()**

 **GS_GetCurListStart**  **GS_GetVidSignal**

GS_GetVidSignal

Returns the signal bit which a GState is using for video synchronization

Synopsis

int32 GS_GetVidSignal(GState* g);

Description

Returns a signal bit mask showing which signal, if any, is being used by a GState object to keep Triangle Engine rendering in sync with double-buffering of bitmaps by the Display Manager. This signal is associated with a GState by calling **GS_SetVidSignal()**.

This signal should normally be the same value which was passed to the Display Folio as the arg for the VIEWTAG_DISPLAYSIGNAL tag, when the View was created.

Arguments

g
: Pointer to a GState object

Return Value

Returns the signal associated with a GState, or 0 if one was not associated with the GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_SetVidSignal()**, **GS_BeginFrame()**, **GS_SendList()**

GS_GetZBuffer

Get the Item number of the bitmap being used as a Z-buffer

Synopsis

Item GS_GetZBuffer(GState* g);

Description

Returns the Item number for the Bitmap being used as a Z-buffer for a GState object. When Triangle Engine commands are rendered through the specified GState object, the Triangle Engine will use this bitmap to hold Z-buffer information. The actual rendered scene is stored in a separate bitmap.

Arguments

g

Pointer to a GState object

Return Value

Returns an Item number for the Bitmap, or an error code if a valid bitmap has not been set for this GState.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_SetZBuffer(), GS_SetDestBuffer(), GS_GetDestBuffer()

 **GS_GetVidSignal**  **GS_Init**

GS_Init

OBSOLETE - Initializes a GState object to a default state

Synopsis

Err GS_Init(GState* g);

Description

This call is OBSOLETE. Use **GS_Create()** instead, to create a GState object.

Arguments

g
Pointer to a GState object

Return Value

Returns 0 for success or a negative error code for failure.

Implementaion

No longer implemented.

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create()

GS_Reserve

Reserve block of memory in GState's current command list buffer

Synopsis

void GS_Reserve(GState* g, uint32 nwords);

Description

Check to ensure that there is enough space in a GState's current command list buffer for the requested number of words. If there isn't enough space, flush the current command list buffer, and if more than one command list buffer was allocated by **GS_AllocLists()**, reserve the requested space in the next available buffer.

Note that this routine will silently fail if the requested number of words is larger than the total size of a command list buffer.

Arguments

g
: Pointer to a GState object

nwords
: Number of words to reserve

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_SendList()**, **GS_SetList()**

 **GS_Init**  **GS_SendList**

GS_SendList

Send a GState's current command list to the Triangle Engine

Synopsis

Err GS_SendList(GState* g);

Description

Send the data in the GState's current command list buffer to the Triangle Engine device driver to be rendered. If more than one command list buffer was allocated when **GS_AllocLists()** was called, the GState will use the next available command list buffer when this function returns.

The Font Folio, 2D Graphics Folio, and 3D Graphics Library build buffers of Triangle Engine instructions using CLT, or the Command List Toolkit. Once the buffer fills up, or gets close to becoming full, **GS_SendList()** is called to flush the buffer and let the Triangle Engine begin rendering.

To support tear-free double-buffering of the frame buffers, **GS_SendList()** will wait on a signal from the video hardware, if one was associated with a GState by calling **GS_SetVidSignal()**. This wait will only occur when the first command list buffer is to be rendered into a frame. To specify that this wait should occur, the user should call **GS_BeginFrame()** whenever a new frame buffer has just been displayed.

Normally, the user application should call g->gs_SendList(g) instead of calling this routine directly, so that if it becomes necessary to debug the command list output, a user function can easily replace **GS_SendList()**. g->gs_SendList() is initialized to use GS_SendList()**g->gs_SendList()** is initialized to use **GS_SendList()** by default when a GState is created.

g->gs_SendList() is called automatically by the GState folio whenever**g->gs_SendList()** is called automatically by the GState folio whenever **GS_Reserve()** cannot allocate enough space in the current command list buffer.

Arguments

g
: Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_Reserve()**, **GS_SetList()**, **GS_SetVidSignal()**, **GS_GetVidSignal()**, **GS_BeginFrame()**

GS_SetDestBuffer

Set a bitmap as the output frame buffer for a GState

Synopsis

Err GS_SetDestBuffer(GState* g, bmItem);

Description

Set a bitmap as the output frame buffer for a GState. When commands are sent to the Triangle Engine by this GState object, the output from the Triangle Engine will go into the specified bitmap.

Arguments

g
: Pointer to a GState object

bmItem
: Item representing a Bitmap object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_GetDestBuffer()**, **GS_SetZBuffer()**, **GS_GetZBuffer()**

 **GS_SendList**  **GS_SetList**

GS_SetList

Set a GState to use a new command list buffer

Synopsis

Err GS_SetList(GState* g, uint32 idx);

Description

Set a GState to use a new command list buffer. These buffers should be allocated with **GS_AllocLists()**. **GS_SendList()** will call this routine automatically to set up the GState to use the next available command list buffer, if more than one buffer has been allocated.

Arguments

g
: Pointer to a GState object

idx
: Index stating which command list buffer should be used. Value should be from 0 ... (numberCmdLists - 1)

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_Reserve()**, **GS_SendList()**

GS_SetVidSignal

Attach a signal to a GState object, for video synchronization

Synopsis

Err GS_SetVidSignal(GState* g, int32 signal);

Description

Associate a signal bit with a GState object. This signal will allow a GState to stay synchronized with the video in a double- buffering scheme.

To correctly enable this feature of GState, allocate a signal by calling **AllocSignal()** before a View is created. Then, use this allocated signal as the argument for the VIEWTAG_DISPLAYSIGNAL tag when a view is created. Last, associate this signal with a GState by calling **GS_SetVidSignal()**. Whenever one bitmap has been fully rendered and sent to the Display Folio, the user should also call **GS_BeginFrame()**, to tell a GState that it needs to wait for this signal before it sends the next command list.

Arguments

g
: Pointer to a GState object

signal
: A signal, created by **AllocSignal()**

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_GetVidSignal()**, **GS_BeginFrame()**, **GS_SendList()**

GS_SetZBuffer

Set a bitmap as the Z-buffer for a GState

Synopsis

Err GS_SetZBuffer(GState* g, Item bmItem);

Description

Set a bitmap as the Z-buffer for a GState. If Z-buffering is enabled for the Triangle Engine, this bitmap will be used to hold the depth information for triangles as they are rendered.

Arguments

g
: Pointer to a GState object

bmItem
: Item representing a Bitmap object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), GS_GetZBuffer(), GS_SetDestBuffer(), GS_GetDestBuffer()

GS_WaitIO

Wait for all pending IO to complete for a GState

Synopsis

Err GS_WaitIO(GState* g);

Description

For a given GState, wait for any active IORequests to complete. This routine is usually used to ensure that the Triangle Engine is not still rendering into a bitmap before that bitmap is displayed.

Arguments

g
: Pointer to a GState object

Return Value

Returns >= 0 for success or a negative error code for failure.

Implementation

DLL call implemented in gstate V29

Associated Files

<:graphics:clt:gstate.h>, System.m2/Modules/gstate

See Also

GS_Create(), **GS_AllocLists()**, **GS_SendList()**

GS_SetZBuffer

CLT_AllocSnippet

Allocate space for the data for a CltSnippet

Synopsis

Err CLT_AllocSnippet(CltSnippet* s, uint32 nWords);

Description

Allocate space for the specified number of words of command list data for a given CltSnippet. The CltSnippet should already have been initialized by calling **CLT_InitSnippet()**.

Arguments

s
: Pointer to a valid, initialized CltSnippet

nWords
: Number of words of memory to be allocated for the CltSnippet

Return Value

Returns 0 for success or a negative error code for failure.

Caveats

If the CltSnippet already has memory allocated for its data, this memory will be left dangling in memory. Use **CLT_FreeSnippet()** to free the memory first.

Implementation

Library call in libclt.a

See Also

CLT_FreeSnippet(), **CLT_InitSnippet()**

CLT_ClearFrameBuffer

CLT_ClearFrameBuffer

Clear frame buffer and/or Z-buffer via CLT cmds

Synopsis

**void CLT_ClearFrameBuffer(GState* gs, float red, float green,
float blue, float alpha, bool clearScreen, bool clearZ);**

Description

Clear the frame buffer and/or Z-buffer via the Triangle Engine. This is done by drawing two large triangles of the specified color. If clearing of the Z-buffer is desired, 0 is written to the whole Z-buffer as well. Note that because the screen clear is done via Triangle Engine commands, the clear doesn't actually occur until the command list is sent to the Triangle Engine.

Arguments

gs
: Pointer to a GState object

red, green, blue, alpha
: Floating point numbers between 0.0 and 1.0, inclusive, which specify the intensities of the R, G, B, and alpha channels, respectively.

clearScreen
: Boolean specifying whether the screen should be cleared. This would be FALSE when the user only wants to clear the Z-buffer.

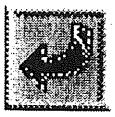
clearZ
: Boolean specifying whether the Z-buffer should be cleared. Note that this value is ignored when a Z-buffer is not being used. Also, when a Z-buffer is being used, this routine will NOT enable Z-buffering if clearZ is set to TRUE. It is the responsibility of the user's code to enable Z-buffering before calling this routine. This is done because normally, the Z-buffer would just be enabled once when the application first starts up, whereas clearing the Z-buffer would be done once per frame.

Implementation

Library call in libclt.a

See Also

GS_Create(), GS_SendList()


CLT_AllocSnippet

CLT_ComputePipLoadCmdListSize

CLT_ComputePipLoadCmdListSize

Compute size of PIP load command list

Synopsis

int32 CLT_ComputePipLoadCmdListSize(CltPipControlBLock* txPipCB);

Description

Computes the size of a PIP load command list snippet, without actually creating the snippet. This routine is most useful when users want to have a PIP load command list be created directly into the command list buffer, and need to know how much space will be needed ahead of time to prevent overflowing the command list buffer.

Arguments

txPipCB

Pointer to a CltPipControlBlock structure. See **CLT_CreatePipCommandList()** for information on how to correctly fill in this structure.

Return Value

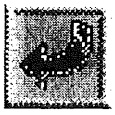
Returns the necessary size, in words, if successful, or a negative error code if the call fails.

Implementation

Library call in libclt.a

See Also

CLT_CreatePipCommandList(), **GS_Reserve()**

 **CLT_ClearFrameBuffer**  **CLT_ComputeTxLoadCmdListSize**

CLT_ComputeTxLoadCmdListSize

Compute size of texture load command list

Synopsis

int32 CLT_ComputeTxLoadCmdListSize(CltTxLoadControlBlock* txLoadCB);

Description

Computes the size of a texture load command list snippet, without actually creating the snippet. This routine is most useful when users want to have a texture load command list be created directly into the command list buffer, and need to know how much space will be needed ahead of time to prevent overflowing the command list buffer.

Arguments

txLoadCB

Pointer to a CltTxLoadControlBlock structure. See **CLT_CreateTxLoadCommandList()** for information on how to correctly fill in this structure.

Return Value

Returns the necessary size, in words, if successful, or a negative error code if the call fails.

Implementation

Library call in libclt.a

See Also

CLT_CreateTxLoadCommandList(), **GS_Reserve()**

 **CLT_ComputePipLoadCmdListSize**  **CLT_CopySnippetData**

CLT_CopySnippetData

Copy data portion of a cmd list snippet to specified location

Synopsis

void CLT_CopySnippetData(uint32 dest, CltSnippet* src)**

Description

Copies data from a CltSnippet to another memory location, updating the dest ptr as it goes. Most often, this routine is used to copy command list snippets into a command list buffer within the GState.

Arguments

dest

Pointer to a pointer to a buffer of 32-bit quantities. On exit from this routine, *dest will be updated to point to just beyond the copied data area, so that a subsequent call to **CLT_CopySnippetData()** can be made without updating the pointer manually.

src

Pointer to a valid, initialized CltSnippet

Implementation

Library call in libclt.a

See Also

GS_Reserve(), **GS_SendList()**, **CLT_InitSnippet()**

 **CLT_ComputeTxLoadCmdListSize**  **CLT_CreatePipCommandList**

CLT_CreatePipCommandList

Creates a cmd list snippet to load a PIP

Synopsis

Err CLT_CreatePipCommandList(CltPipControlBlock* txPipCB);

Description

Creates a command list snippet to load the specified block of memory as a PIP, or Pen Index Palette, for indexed textures to use. The fields of the CltPipControlBlock should be filled in as follows:

pipData
: Pointer to the PIP data in RAM

pipIndex
: Starting index of the data to load

pipNumEntries
: Number of entries in the data to be loaded

Arguments

txPipCB
: Pointer to a CltPipControlBlock structure. On entry, txPipCB should be filled in as described above. On exit, the field pipCommandList will be filled in with a command list snippet that can be copied into a command list buffer when the PIP load needs to occur. See **CLT_CopySnippetData()** for more details.

Return Value

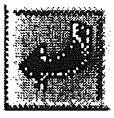
Returns 0 for success or a negative error code for failure.

Implementation

Library routine in libclt.a

See Also

CLT_CopySnippetData(), **CLT_ComputePipLoadCmdListSize()**

 **CLT_CopySnippetData**  **CLT_CreateTxLoadCommandList**

CLT_CreateTxLoadCommandList

Creates a cmd list snippet to load a texture

Synopsis

Err CLT_CreateTxLoadCommandList(CltTxLoadControlBlock* txLoadCB);

Description

Creates a command list snippet to load the specified texture into the Triangle Engine's TRAM, or Texture RAM. For uncompressed textures which do not require runtime carving, the data for the one or more levels of detail is transferred as one large block of contiguous RAM, for optimal performance. For indexed textures, the PIP must be loaded separately, by calling **CLT_CreatePipCommandList()**. The fields of the txLoadCB field should be filled in as follows:

textureBlock
: Pointer to a CltTxData block. This block contains data about the actual texels, such as num LODs, texture format, etc.

numLOD
: Number of LODs (Levels of Detail) in texture

XWrap
: Wrap mode in X-direction (0=Clamp, 1=Tile)

YWrap
: Wrap mode in Y-direction (0=Clamp, 1=Tile)

XSize
: Width of area to load. Normally equal to the minX field of the textureBlock structure

YSize
: Height of area to load. Normally equal to the minY field of the textureBlock structure

XOffset
: Left edge of sub-texture to load. For non-carved textures, this will normally be 0.

YOffset
: Top edge of sub-texture to load. For non-carved textures, this will normally be 0.

tramOffset
: Offset into the 16k TRAM where the texture should be placed. For uncompressed, uncarved textures, this value must be 32-bit aligned.

tramSize
: Size texture will require in TRAM.

Arguments

txLoadCB
: Pointer to a CltTxLoadControlBlock structure. On entry, txLoadCB must be filled in as described above. On exit, the field lcbCommandList will contain a command list snippet to load and use the specified texture. Additionally, the useCommandList can be used to just re-use the texture, assuming that it is already loaded. The useCommandList is most useful when multiple textures reside in the TRAM at once, and the user data needs to alternate between these different textures.

Return Value

Returns 0 for success or a negative error code for failure.

Implementation

Library routine in libclt.a

See Also

CLT_CopySnippetData(), CLT_ComputeTxLoadCmdListSize(), CLT_CreatePipCommandList()

 CLT_CreatePipCommandList  CLT_FreeSnippet

CLT_FreeSnippet

Free memory allocated for a CltSnippet's data

Synopsis

void CLT_FreeSnippet(CltSnippet* s);

Description

Frees the memory allocated to a CltSnippet's data area. The memory is only freed if it was allocated for the snippet. That is, there are times when several snippets may all share the same data area, such as when one is only supposed to represent a smaller portion of another. In that case, calling **CLT_FreeSnippet()** on the smaller sub-snippet will not cause the smaller portion of memory to be freed from the larger memory block.

Arguments

s

Pointer to a valid, initialized CltSnippet

Caveats

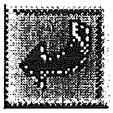
In the case where a snippet is using memory from a larger snippet, and **CLT_FreeSnippet()** is called to free the memory used by the larger snippet, there is no way to mark the smaller snippet that is sharing the memory as freed. This could result in a CltSnippet pointing to garbage memory.

Implementation

Library call implemented in libclt.a

See Also

CLT_AllocSnippet()

 **CLT_CreateTxLoadCommandList**  **CLT_InitSnippet**

CLT_InitSnippet

Initializes a CltSnippet structure

Synopsis

void CLT_InitSnippet(CltSnippet* s);

Description

Initializes a CltSnippet structure. The structure must be created before calling this routine, either by calling **AllocMem()** or by just declaring a variable of type CltSnippet. Once initialized, a CltSnippet can be used in many other CLT calls.

Note that initializing a CltSnippet does not allocate any memory for the snippet's data space. To allocate that memory, call **CLT_AllocSnippet()** after calling **CLT_InitSnippet()**.

Arguments

s

Pointer to a valid CltSnippet

Implementation

Library call in libclt.a

See Also

CLT_AllocSnippet(), **CLT_CopySnippet()**, **CLT_CopySnippetData()**, **CLT_FreeSnippet()**, **AllocMem()**

 **CLT_FreeSnippet**  **CLT_SetSrcToCurrentDest**

CLT_SetSrcToCurrentDest

Set the Dest Blender so that src blends will occur with cur frame buffer

Synopsis

void CLT_SetSrcToCurrentDest(GState* g);

Description

This routine adds some commands to the current command list buffer of a GState which set the destination blend source buffer equal to the current destination frame buffer. This is usually most useful when trying to make objects appear translucent. To achieve a translucent effect, an object gets blended with the current frame buffer in the destination blender of the Triangle Engine.

Note that this routine does not actually enable source blending, nor does it set up the blend operation. It merely sets the following CLT attributes: DBSRCBASEADDR, DBSRCXSTRIDE, DBSRCOFFSET, DBSRCCNTL.

Arguments

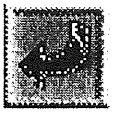
g
: Pointer to a GState object

Implementation

Library call in libclt.a

See Also

GS_Create(), **GS_GetDestBuffer()**



CLT_InitSnippet

CltEnableTextureSnippet

Global var. used to enable texturing

Description

The CltSnippet **CltEnableTextureSnippet** can be inserted by the user into the current command list buffer when texturing needs to be

enabled. This CltSnippet contains the following commands:

1. Enable the TEXTUREENABLE bit of the TXTADDRCNTL register.

To actually use this CltSnippet, call **CLT_CopySnippetData()**.

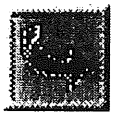
Note that if the CltSnippet **CltNoTextureSnippet** is used to disable texturing, using **CltEnableTextureSnippet** will not completely restore state. It will also be necessary for the user's code to set the COLOROUT and ALPHAOUT fields of the TXTTABCNTL register, to either output the texture's colors, or to a blend operation between the primitive's color and the texture's color.

Implementation

Global variable exported in library libclt.a

See Also

CltNoTextureSnippet, **CLT_CopySnippetData()**



CltNoTextureSnippet

CltNoTextureSnippet

Global var. used to disable texturing

Description

The CltSnippet **CltNoTextureSnippet** can be inserted by the user into the current command list buffer when texturing needs to be disabled.

This CltSnippet contains the following commands:

1. Sync the Triangle Engine 2. Disable the TEXTUREENABLE bit of the TXTADDRCNTL register 3. Set the TXTTABCNTL COLOROUT to PRIMCOLOR, and the ALPHAOUT to PRIMALPHA

To actually use this CltSnippet, call **CLT_CopySnippetData()**.

Implementation

Global variable exported in library libclt.a

See Also

CltEnableTextureSnippet, **CLT_CopySnippetData()**

CltEnableTextureSnippet

Index

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W